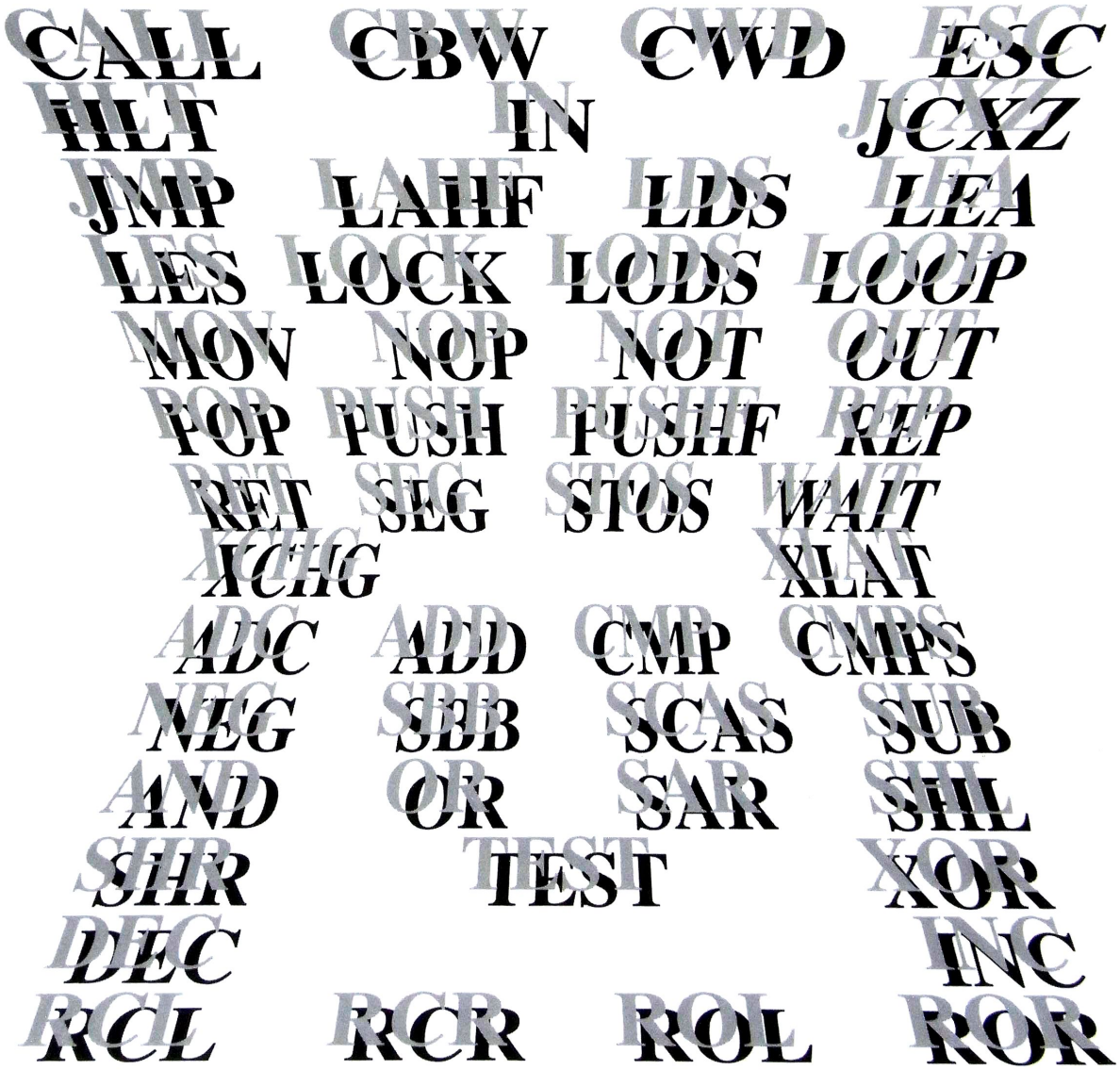
I/O 別冊

PC-9801用2HDフロッピーディスク各1枚付

86 Assembler

初心者が短時間で上達するための 86アセンブラ入門

mini ZZ386
PC-9801用逆アセンブラ



出海　勝富著

工学社

CONTENTS

●8086アセンブラ入門

●まずニモニックを少しだけ覚えましょう

●アセンブラ疑似命令を少し

●逆アセンブラ『ZZ386』を使う

●実行ファイルの種類

●プログラムの実行開始と終了

●サンプル・プログラム解説

●MS-DOS Version5.0
ファンクション・リクエスト（Int21h）一覧

I/O 別冊

86 Assembler

初心者が短時間で上達するための 86アセンブラ入門

mini ZZ386
PC-9801用逆アセンブラ

CALL CBW CWD ESC
HLT IN JCXZ
JMP LAHF LDS LEA
LES LOCK LODS LOOP
MOV NOP NOT OUT
POP PUSH PUSHF REP
RET SEG STOS WAIT
XCHG XLAT
ADC ADD CMP CMPS
NEG SBB SCAS SUB
AND OR SAR SHL
SHR TEST XOR
DEC INC
RCL RCR ROL ROR

出海　勝富 著

工学社

86 Assembler

初心者が短時間で上達するための 86アセンブラ入門

mini ZZ386

PC-9801用逆アセンブラ

まえがき

初心者が短時間で上達するためのアセンブラ入門

《この本の使い方》

■「アセンブラ入門」はよく目にするタイトルです．しかし，いきなり多くの情報，資料が目の前に現われても，何から手を付ければ良いのかとまどってしまうのが現実だと思います．

本書では最初に「覚えなければならないこと」を説明します．しかし，すべてを覚えないと次に進めないわけではありません．100点満点を取らなくても2年生に進級できるようなものです．合格ラインは30～50点と考えてください．それらの知識はプログラムが一応書けるようになってから少しずつ補えばよいのです．

■次に，「他人のプログラムから参考にすべきこと」を説明します．よく，「プログラム上達の近道は多くのプログラムを読むこと，書くこと」だと言われますが，ここでは「読むこと」を中心に述べます．

最近はソース・リストの付いたプログラムも流布していますが，多くは実行形式だけのプログラムです．ここでは対話形式の逆アセンブラ『ZZ386ミニ』を使って，それらのプログラムを「読む」方法についても述べてあります．

■この本を読んでいて「私にも書けそうだ」あるいは「何か書いてみたい」と感じたらプログラムを書き始めてみましょう．テーマはあなたが興味のあることを選んでください．

いきなり「データベースを作ろう」，「ワープロを作ろう」という大胆な人はいないと思いますが，あなたの現在の能力が100とすれば150くらいのテーマを選ぶのがベストではないかと思います．書いてるうちに上達するからです．もちろん，この本も並行して読んでください．

《この本を読むための準備》

この本を読むのにも少しばかり予備知識が必要です．それにパソコンと，アセンブラなどのソフトも必要です．パソコンはMS-DOSが動作すればOKですが，困ったときに教えてくれる人がたくさんいるPC-9801あたりが無難でしょうか．

必要なソフト

①まず，オペレーティング・システム（OS）のMS-DOSが必要です．これは機種によって違い，各機種ごとにパソコン・メーカーから売られています．

②また，あなたが書いたプログラムをパソコンで動作する形（実行ファイル）に変換するアセンブラが必要です．これは有名なところではMASM（マイクロソフト），TASM（ボーランド），OPTASM（SLR）などがあります．まだ持っていないのでしたら，性能，コスト，マニュアルの出来からTASMをお勧めします．

③そしてプログラムを書くためにエディタが必要です．有名なところではMIFES，Vzエディタなどがあります．

以上をまとめると，最低，

パソコン ＋ MS-DOS ＋ アセンブラ ＋ エディタ

が必要ということです．

ただし，これらについての知識はとりあえず多くは必要ありません．

《必要なMS-DOSの知識》

MS-DOSのコマンド**copy**，**dir**などが使えて，エディタが取りあえず操作できればけっこうです．徐々に熟達していきましょう．全然ダメなときは少なくともキーボードからプログラム名を打ち込んで動かす「起動」ができるまでにはなってください．

Copy
コピー

DIRectory
ディレクトリ
「ディーアイアール」と読むことが多い．

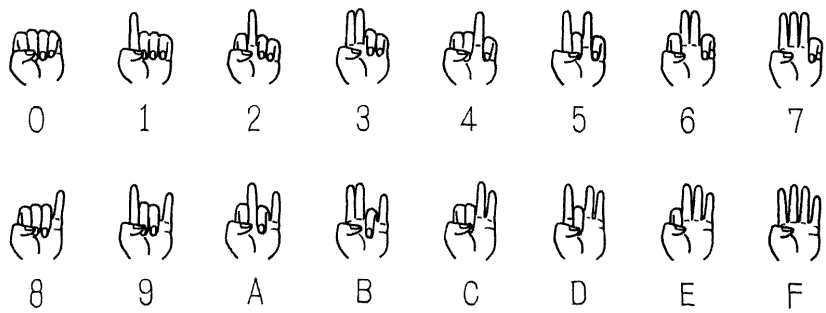
「アセンブラ入門」に必要なもの

パソコン	PC-9801などMS-DOSが動作するもの
MS-DOS	各機種（パソコン）にあったもの． 各メーカーから販売されている． 最新バージョンはVer.5
エディタ	MIFES，Vzエディタなど 使う機種にあったものが必要
アセンブラ	TASM，MASM，OPTASMなどの8086用アセンブラ

指で数える16進

私たちは10進数の世界で生活しているので，2，16進数で計算することには慣れていませんが，プログラムではビットを扱うことがよくありますビットとは2進数そのものです．

「2進数⇔16進数」の変換は頻繁に要求されます．こんなときいちいち紙に書いていたのでは効率が良くありません．子供が算数を覚えるとき指折って数えますが，これを応用した16進数の数え方を身に付けましょう．5～10分で覚えられますのでやってみてください．とても重宝しますよ．



CONTENTS

まえがき……02
第1部　8086アセンブラ入門……09
1　プログラムの立場……10
実行
ハードウェアを直接操作する
システム・コールを使う
ソフトウェア割り込み
2　86系CPUのレジスタ構成……12
汎用レジスタ
セグメント・レジスタ～面倒なセグメントのお話～
インデックス・レジスタ
アドレッシング・モード
3　システム・コール（割り込みサービス），BIOSの利用……16
8086の割り込み
割り込みサービスの分類
int 21h
その他の20h番台の割り込みサービス
BIOSコール
4　アセンブルとは……20
ソース・ファイル
オブジェクト・ファイル
リンク
5　システム（パソコン）の設定……21
CONFIG.SYSの詳細
AUTOEXEC.BAT の詳細

第2部　まずニモニックを少しだけ覚えましょう……27
1　機能から見たニモニック（命令）の分類……28
2　データのコピー命令……29
メモリの指定方法＝アドレッシング・モード
直接アドレッシング
レジスタ間接アドレッシング
セグメント・プリフィックス
3　算術演算命令……31
加算命令
ADD,ADC
INC
AAA(Ascii Adjust for Addition),DAA(Decimal Adjust for Addition)
減算命令
SUB,SBB
DEC
乗算命令
除算命令
比較命令
4　論理演算命令……35
AND（論理積命令）
NOT（否定命令）
OR（論理和命令）
XOR（排他的論理和命令）
TEST

5 ストリング命令…… 37
ストリング・プリミティブ命令
MOVS
LODS
STOS
SCAS
CMPS
比較命令の方向
CLDとSTD
リピート・プリフィックス命令
REP
REPE,REPZ
REPNE,REPNZ
6 ジャンプ，コール命令…… 42
CALL命令
RET命令
JMP命令
条件ジャンプ命令
ループ命令
7 プロセッサ制御命令…… 48
8 入出力命令…… 48
9 割り込み命令…… 49
10 シフト，ローテート命令…… 49
キャリーフラグを含めてリング状のパターン（ローテート）
右に移動
左に移動
キャリーフラグを含まずリング状のパターン（ローテート）
左に移動
右に移動
直線状のパターン（論理シフト）
左に移動
右に移動
直線状のパターン（算術シフト）
右に移動

第3部 アセンブラ疑似命令を少し…… 55
1 COMファイルとEXEファイル…… 56
疑似命令「SEGMENT」
SEGMENT疑似命令の使い方
2 複数のソース・ファイルから構成されるCOMファイル…… 66
3 複数のソース・ファイルから構成されるEXEファイル…… 71
4 プログラムの単位＝プロシージャ…… 77
ローカル・ラベル
5 数式，定義のディレクティブ…… 79
EQUと「=」
6 ラベル関連…… 80
PUBLIC
EXTRN
GLOBAL
LABEL命令
LOCAL
7 マクロ…… 84
8 インクルード・ファイル…… 86

第4部　逆アセンブラ『ZZ386』を使う…… 89
1　プログラム上達の近道…… 90
2　『ZZ386ミニ』の仕様…… 91
3　『ZZ386ミニ』の使い方の概略…… 92
『ZZ386ミニ』のインストール
プロジェクト・ファイルの作成
新しくプロジェクト・ファイルを作成
プロジェクト・ファイルを選択
プロジェクト・ファイルの設定変更
ZZ386起動時に
オプション・メニューから（ZZ386実行中）
プロジェクト・ファイルの諸設定
逆アセンブルの実行
メイク（自動解析）
逆アセンブル結果の出力
ファイル形式のオプション
出力デバイスの選択
エリアの指定
出力ファイル名の確認
ZZ386の終了
アセンブルの実行（プログラムの再現）
4　『ZZ386』のコマンド……106
カーソル移動，スクロールなどの画面操作
ヘルプの表示
コマンドのキャンセル
カーソル移動とスクロール
ファンクション・キーに割り当てられている主なコマンド
「ZZ386」の終了
オプション・メニュー
コメント
ラベル
ジャンプ
サーチ
セグメント値
クロス・リファレンス
逆アセンブルの出力
自動解析
簡単な変更コマンド（主にマニュアル操作）
カーソル行の属性変更
ブロック単位の属性変更
キャラクタ部分の表示
セグメント評価値の表示非表示
プロシージャの変更
ロード値，ジャンプテーブルの表示，変更
カーソル行をタグにセット，削除

第5部　実行ファイルの種類…… 141
1　COMファイル，EXEファイル…… 142
セグメントレジスタの初期化
2　デバイス・ドライバ……145
デバイス・ヘッダ
ストラテジ・ルーチン
割り込みルーチン
3　常駐型プログラム……149
通常のプログラム　TIMEA.COM
常駐型のプログラム　TIMES.COM
TIMESを常駐して終了する
TIMESを解放して終了する
時刻を表示する

第6部　プログラムの実行開始と終了……157
プログラムの初期化部分
コマンドライン・オプションなどの受け渡し
終了部分
終了
常駐終了

第7部　サンプル・プログラム解説……165

付録 MS-DOS Version 5.0ファンクション・リクエスト（int 21h）一覧… 171

添付ディスクの使い方…… 199

最後に…… 202

INDEX…… 204

●本書の添付ディスクに収録されているプログラムは，運用者本人の判断で運用にあたってください．プログラム上のいかなるバグ，操作上のいかなるミス，あるいはプログラムそのものによる金銭的損失について，（株）工学社，（株）マイクロ・オーグ，ならびに筆者はいっさい責任を負えませんので，あらかじめご了承ください．
●本書の記載内容，添付ディスクの内容に関して，（株）マイクロ・オーグへの直接のお問い合わせはご遠慮ください．これらに関するご質問は，（株）工学社宛てにお願いいたします．

下記の名称は右に示す各社の商標です．

ZZ386	マイクロ・オーグ
MS-DOS MS-Windows Macro Assemblar	マイクロソフト
Turbo Assemblar Turbo C	ボーランド
Vz・エディタ	ビレッジセンター

CBW CWD ESC
IN JCXZ
HF LDS LEA
CK LODS LOOP
OP NOT OUT
SH PUSHF REP
EG STOS WAIT

第1部　8086アセンブラ入門

この「アセンブラ入門」では，最初はできるだけ覚えることを少なくして必要最低限のことだけを覚え，少しでも早くアセンブラ・プログラムが書けるように構成されています．「まず，ビルの鉄骨を組み上げて，内装は徐々にやっていこう」というスタイルです．美しい内装にしたかったらショウルームでも見にいってセンスを磨けばいいじゃないですか．

XCHG XLAT
DD CMP CMPS
BB SCAS SUB
OR SAR SHL
TEST XOR
INC
CR ROL ROR

1　プログラムの立場

余計なことかも知れませんが，パソコンについて考えてみましょう．「バカバカしい」と思う人は次の節へ進んでください．

パソコンのプログラムはディスク（ハードディスク）などにファイルとして記録されています．キーボードからコマンド（ファイル名＝プログラム名）を入力するとプログラムはパソコンのメモリに読み込まれて実行されます．

どうやってディスクからメモリに読み込まれ実行が始まるのかは，とりあえず考えないことにしましょう．それはMS-DOSにまかせておきましょう．

実行

読み込まれたプログラムの立場になって考えてみると「実行される」とは，CRT，プリンタ，ディスク，通信を操作することです．

これらを操作するには2つの方法があります．

ハードウェアを直接操作する

ひとつは直接ハードウェアを操作するやり方です．プログラムのスピードは速くなる可能性（？）がありますが，プログラムを書くために非常に多くの資料が必要です．

また，公開されていなかったり入手困難な情報もたくさんあります．そして重要なのは互換性です．このようにして書かれたプログラムはまず互換性がありません．PC-9801用に書かれたプログラムはFM-Rシリーズでは動きませんし，逆にFM-R用に書かれたプログラムはPC-9801では動きません．CRTやプリンタの操作方法が違うからです．

システム・コールを使う

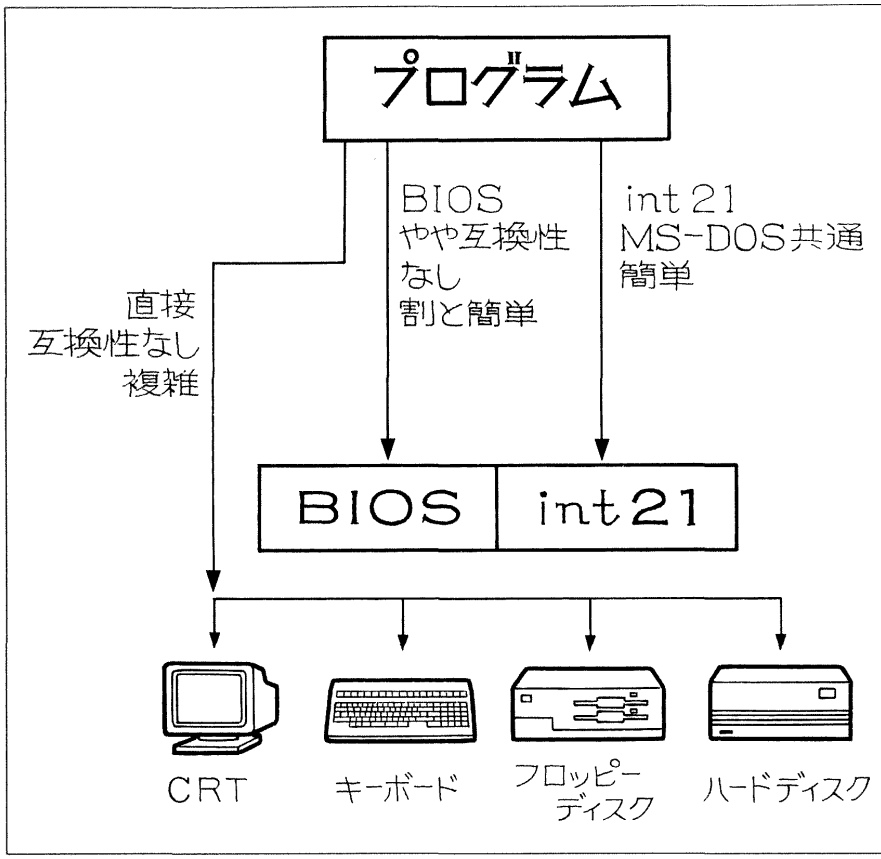
システム・コール
System Call

もうひとつはシステム・コールを利用する方法です．システム・コールはMS-DOSがもっている画面の表示，キーボードからの入力，ディスクのリード/ライトなどのよく使うプログラムの集まりです．

図を見てください．ハードウェアを直接操作するプログラムは違う機種では動きません．専用のプログラムを用意する必要があります．同じMS-DOSなのにこれでは不便ですね．

一方，システム・コールを使ったプログラムはPC-9801でもFM-Rでも動作します．このほうが便利ですね．システム・コールを使うとプログラムを書くのがずいぶん簡単になります．他機種でも動作するのはシステム・コールがハードウェアの違いを吸収してくれるからです．

イラスト1



システム・コールと似たものにBIOSがあります．BIOSとは「Basic Input Output System」の略で，システム・コールと同じような働きをしますが，BIOSは機種により内容（働き）が違います．BIOSはBIOSはPC-9801にはPC-9801のBIOSがあり，FM-RにはFM-RのBIOSがあります．

*

システム・コールとBIOSについてはあとで詳しく説明します．

ソフトウェア割り込み

システム・コールやBIOSは8086の「ソフトウェア割り込み」を使って呼び出されます．命令は，

```
int 21h
int 10h
```

などです．数字の部分は00h～FFh（16進）が指定できます．10進数では0～255になります．

ファンクション・コール
Function Call

この数字が「21h」のときを特に「MS-DOSファンクション・コール（システム・コール）」または「int21ファンクション」と呼んでいます．これらはすべての機種で共通して使えます．IBM PC/AT，AX，DOS/V，J-3100，PC-9801，FM-Rどれでも使えます．

この他にも20h，23h，24h，25hなどはMS-DOSが提供するサービスで，各機種共通で使えます．

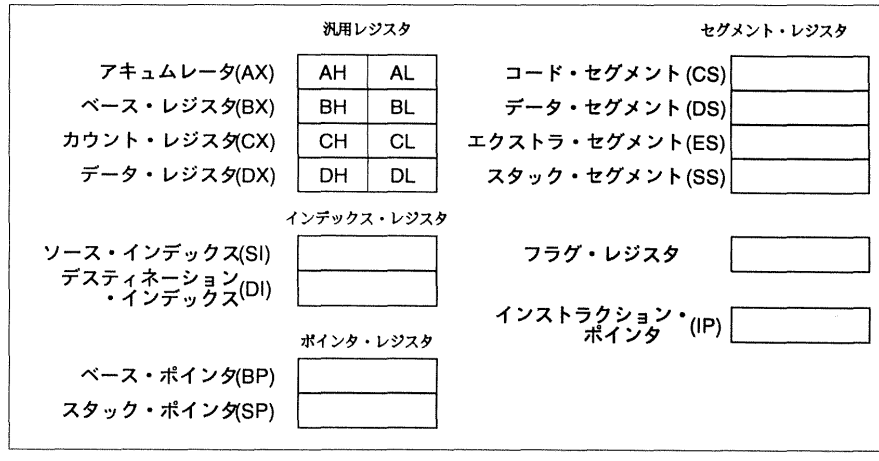
それ以外は基本的に機種により違います（この部分がBIOSです）．IBM PC/ATとPC-9801はまったく違うと考えたらよいでしょう．AX，DOS/V，J-3100などはIBM PC/ATをもとに日本語処理サービスなどが拡張されています．

そのため，できるだけ「システム・コール」を使い，それでは実現できないときだけBIOSを使うか直接ハードウェアを操作するのが一般的です．

BIOSの使い方も将来は覚える必要がありますが，ここでは「BIOS」，「intサービス」，「int21ファンクション」というキーワードだけ記憶しておいてください．

2　86系CPUのレジスタ構成

全レジスタの図



ビット
Bit

8086には16ビット汎用レジスタ（4個），16ビット・インデックス・レジスタ（5個），セグメント・レジスタ（4個），16ビット・フラグ・レジスタ（１個）があります．

汎用レジスタ

汎用レジスタはその名のとおり色々な用途に使われるレジスタです．

AX，**BX**，**CX**，**DX**がありますが，ハイバイト，ローバイト別々にも操作できるようになっています．

バイト
Byte

ビット，バイト，ワード，ダブル・ワード

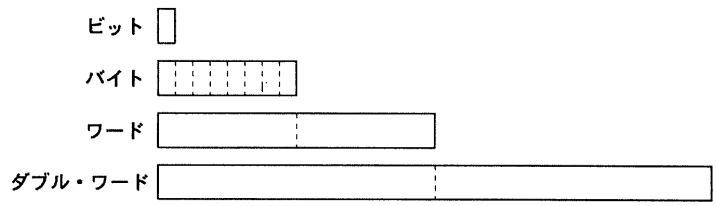
ビットはコンピュータの扱える最小単位で，0か1の2つの値のみが表現できます．「1バイト＝8ビット」となっています．これは「1m＝100cm」と同じで決まりですから，覚えるより仕方ありません．

1ビットでは2つの値しか表現できませんが，1バイトでは0～255の256種類の値が表現できます．これで通常の英数字（キーボードに印刷されてあるアルファベット，数字，記号）なら充分に表現できます．この割り付けを「ASCII（アスキー）コード」といいます．

ASCII
American Standard Code for Infomation Interchange の略

パソコンは8ビットCPU，16ビットCPU，32ビットCPUと高機能になってきましたが，これは1回に扱えるデータの大きさを表わしています．16ビットCPUでは1度に2バイト（＝16ビット）のデータを処理します．

この単位は「バイト・データ」，「ワード・データ」，「ダブルワード・データ」などのデータの種類を表わすのにも使われます．当然，16ビットCPUにもバイト・データを扱う命令があります．



セグメント・レジスタ～面倒な「セグメント」のお話～

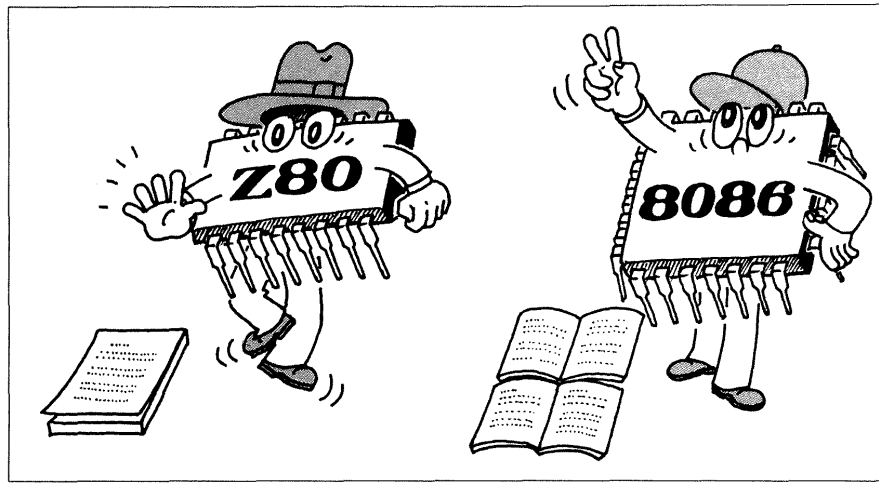
セグメント・レジスタ
Segment Register

よく，「セグメント」は86系CPUの欠点のように言われます．Z80のころからアセンブリ言語でプログラムを書いている人にも，これが嫌で「もうアセンブラはやめた」という人もいます．このあたりが大きな理由でモトローラの68000のファンになった人も多いように感じます．しかし，最近プログラムを始めた人にはそのような不満はあまりないようです．86系が主流なので素直に受け入れるのでしょうね，私はそれが正常な感覚だと思います．

Z80のころでも“バンク切り換え”などは普通のことでしたし，それがそんなに悪いことなんでしょうか．

言葉だけで説明すると難解になるので，絵を使って説明しましょう．

イラスト2



本を読む場合を考えてください．左と右の2ページを同時に見ることができますね．Z80では1ページだけしか見れません．この1ページの中にコード，データ，スタックが全部入っています．日めくりをはがすのが，ちょうどバンクを切り換える感じです．

これに対して86系のCPUは同時に4ページが開ける本のようなものです（そんな本が本当にあれば便利なのですが）．それぞれのページはコード，データ，スタック，特別なデータとなります．

いつも4ページ分開いていると考えれば諦めもつき，抵抗なく受け入れられるのではと思います．そしてプログラムを書くときはコード，データ，スタックの分類をちょっと意識するようにすれば，むしろプラスに転化できるのではないでしょうか．そしてコード部分に自分自身を擬人化してみるのです．

8086ではこれらの4ページを指すのに**CS**，**DS**，**SS**，**ES**の4つの「**セグメント・レジスタ**」を使います．

インデックス・レジスタ

インデックス・レジスタ
Index Register

インストラクション・ポインタ
Instruction Pointer

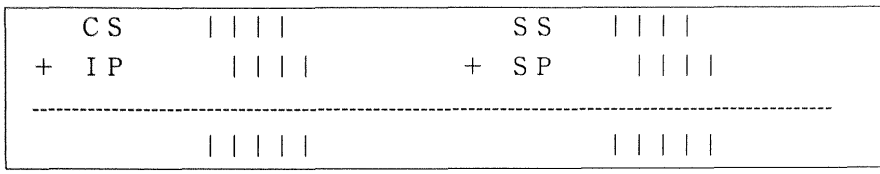
スタック・ポインタ
Stack Pointer

インデックス・レジスタの中でも**IP**（インストラクション・ポインタ）と**SP**（スタック・ポインタ）は特殊なレジスタです．

これらのレジスタはプログラムを書くときにはあまり意識されません．なぜなら，プログラムの実行にしたがって自動的に増減されるからです．特にIPは直接操作することはできません．プログラムの実行，あえていえばジャンプ命令で変更することができます．

これらはそれぞれ現在実行中のプログラムの位置とスタックの位置を指しています．

次のようにそれぞれ「**CS+IP**」，「**SS+SP**」の形で場所を指し示しています．



データを指すための補助的なレジスタ，その名の通りインデックスするためのレジスタには，BP（ベース・ポインタ），SI（ソース・インデックス），DI（デスティネーション・インデックス）があります．

ベース・ポインタ
Base Pointer

ソース・インデックス
Source Index

デスティネーション・インデックス
Destination Index

アドレッシング・モード

メモリの特定の場所を指す方法を「アドレッシング・モード」と言いますが，詳細はp.30を参照してください．ここではその一番複雑な方法のみを説明しましょう．

アドレッシング・モード
Addressing Mode

①**DS**，**CS**，**SS**，**ES**（特に指定しないときはDS）のいずれか
②**BX**または**BP**レジスタ
③**SI**または**DI**レジスタ
④ディスプレースメント（1または2バイトの変位）

各①，②，③，④のグループのいずれか１つを組み合わせることができます．
たとえば，

```
DS:[BX+SI+1200H]
```

のようになります．

①のグループは省略できませんが（省略するとDSの意味になる），②，③，④のグループは省略できます．

例外として，

```
[BP]
```

だけはありません．しかし，

```
[BP+00]
```

とすれば同じ意味になります．

3 システム・コール（割り込みサービス），BIOSの利用

システム・コール
System Call

MS-DOS上で動作するプログラムを書くときにはシステム・コールを使うことは必須となります．したがってこれらも必ず覚えなければなりませんが，これらすべてを一度に覚えることは大変なので，少しだけ覚えましょう．

あるいは，「こういうものがある」ということだけを気に止めておいてください．

必要なときに参考書や資料を見ればすむのですから．

8086の割り込み

割り込み
Interrupt

8086の割り込みは，その番号00hからFFhまでの256種類が使えます．

それぞれの番号の割り込みが発生すると，「0000:0000」番地から番号順に各4バイトごとに記録されているアドレスにジャンプします．

8086の割り込みでまず注意しなければならないのは，外部あるいはCPUから発行される割り込みと，ソフトウェアが発行する割り込みが同レベルで処理されることです．

ファンクション・コール
Function Call

Z80，特にCP/Mから始めた人はちょっと違和感を感じると思います．CP/Mでいうところの「ファンクション・コール」は，あくまで**CALL**命令でしたが，ここは単純に「**CALL**の代りに**INT**を使っている」と考えてください．

ユーザー（私たち）がソフトウェア割り込みを発行して利用可能なプログラムを指して，「BIOS」または「システム・コール」と呼びます．＊印の付いている番号が通常利用できる割り込み（BIOS，システム・コール）です．

その他の割り込みはハードウェアが発行する割り込みです．

これらの割り込み処理プログラムもユーザーが作ったプログラムと置き換えて，より高度な処理をさせることもできます．

20h番台の割り込み処理プログラムは機種が違っても同じ機能と考えても問題ありません．PC-9801，IBM PC/AT，FM-R，AX，J-3100など，MS-DOSを使っている機種なら互換性があります．これらのINTコールだけで処理されているプログラムはどの機種でも動くでしょう（たぶん！）．

図　PC-9801での例

割り込み番号	内　容
00	0除算エラー
01	シングル・ステップ
02	NMI
03	INT3（デバッガ用予約）
04	オーバーフロー
05	COPYキー
06	STOPキー
07	インターバル・タイマ
08	タイマ
09	キーボード入力割り込み
0A	V-Sync
18*	キーボード，CRT　BIOS
19*	RS-232C BIOS
1A*	プリンタBIOS
1B*	ディスクBIOS
1C*	カレンダ，時計BIOS
20*	プログラムの終了（最近はあまり使わない）
21*	システム・コール
22	プログラムの終了アドレス（プログラム側からはあまり使わない）
23	CTRL-Cの処理
24*	致命的ディスク・エラー処理
25*	アブソリュート・ディスク・ライト
26*	アブソリュート・ディスク・リード
27*	プログラムの常駐終了（最近はあまり使わない）
28	アイドル割り込み（キー入力待ちのとき発行される）
29*	コンソール高速文字出力
DC*	拡張機能割り込み（BIOS）

CRT，ディスク，プリンタなどのBIOSはまず互換性がありません．最近の画面のきれいなソフトや通信ソフトなどがそれぞれの機種専用になっているのは，それらの機能が20h番台の割り込みサービスだけでは実現できないからです．

MS-Windowsはそれらも含めて共通にしようという試みでもありますが，あまり上手くいっているとはいえないのかな？

割り込みサービスの分類

「割り込みサービス」はその呼び方も混乱しています．ここでは割り込み（**INT**）を発行してBIOS（システム・コール）を利用することを，一般的に「割り込みサービス」と呼ぶことにします．

20h番台の割り込みは各機種とも共通なので「**int21h**」，「**int24h**」などと呼ぶことにします．

それ以外は「98int18」とか「98キーボードBIOS」と呼ぶことにします．

int 21h

int 21hは機能も多く，使用頻度も高いものです．**AH**レジスタにファンクション（機能）番号を入れ，必要ならばその他のレジスタにパラメータや補助機能番号などを入れて割り込みの**21h**を発行します．

使い方

①機能番号を**AH**レジスタにセットする．
②必要ならば補助機能番号を**AL**レジスタにセットする．
③必要なパラメータをレジスタにセットする．どのレジスタにセットするかは機能番号によって違う．
④「**int 21h**」を実行する．
⑤機能番号によっては，戻り値がいずれかのレジスタにセットされる．

［例］キーボードから1文字入力

```
mov   ah,07h
int   21h
;     al <-- key board
```

詳細は付録の「int21h一覧」を見てください．

その他の20h番台の割り込みサービス

これらの割り込みはプログラム側から使うものと，ユーザーが用意したものを置き換えて登録したりしてMS-DOSからのみ呼び出されるものに分類できます．

プログラム側から使う割り込みには次のものが分類できます．

20	プログラムの終了（最近はあまり使わない）
25	アブソリュート・ディスク・ライト
26	アブソリュート・ディスク・リード
27	プログラムの常駐終了（最近はあまり使わない）
29	コンソール高速文字出力

これらには「**int 21h**」のようなAHの値（機能番号）による分類はありません．必要なパラメータをセットして割り込みを発行します．

MS-DOSから呼び出されるものは次のようなものです．

23	CTRL-Cブレークの処理
24	致命的ディスク・エラー処理
28	アイドル割り込み（キー入力待ちのとき発行される）

通常，プログラムを実行中にCTRL-Cが押されるとMS-DOSは「**int 23h**」を実行し，呼び出された「**CTRL-Cブレークの処理プログラム**」は実行中のプログラムを終了し，ディスクをリセットします．

ところが，エディタなどではCTRL-Cを押したぐらいでプログラムが終了しては困るわけです．

このようなとき，元の処理プログラムと，プログラマーが用意したプログラムを置き換えるのです．

「**致命的ディスクエラー処理**」も同じです．「ドライブの準備が出来ていません．」などのメッセージが出て画面が乱れると困るときなどに置き換えます．

「**アイドル割り込み**」にはもともとは何もしないプログラム（**IRET**命令）が登録されています．ここに何か別の処理をするプログラムとか時計を表示するプログラムを登録し，いわば疑似的なマルチタスクを実現するのに使われます．ただし，この機能はNECのMS-DOSのマニュアルには記載されていません．

22	プログラムの終了アドレス（プログラム側からはあまり使わない）

ちょっとおもむきが違うのは「**int 22h**」です．これはDOSが内部的に参照したりプログラム側からもまれに参照されることがありますが，まず必要ないでしょう．

2F	多重割り込み

そしてとても複雑で，MS-DOSのバージョンアップの度にどんどん拡張されているのが「**int 2Fh**」です．これについてはここでは触れません．

BIOSコール

20h番台以外の割り込みを発生し，システムに用意されたプログラムを利用することを一般にこう呼んでいます．これらは機種により違い，まったく互換性がないと考えましょう．

使い方は「**int 21h**」と良く似ています．

使い方

①機能番号をレジスタにセットする．セットするレジスタはBIOSによって違う．

②必要ならば補助機能番号をセットする．どのレジスタにセットするかはBIOSによって違う．

③必要なパラメータをセットする．どのようにセットするかはBIOSによって違う．

④「int ○○h」を実行する．

⑤BIOSによっては戻り値がレジスタなどにセットされる．

［例］キーボードのシフト状態を知る（PC-9801）

```
mov   ah,02h
int   18h
;     al <-- key board sift status
```

詳細は各機種ごとに技術資料を参照してください．

4　アセンブルとは

アセンブル
Assemble

念のために，これからの話の前提として「アセンブル・プログラム開発」の全体的なイメージを書いておきます．

ソース・ファイル

ソース・ファイル
Source File

最初にアセンブリ言語で書かれたソース・プログラムをエディタを使って作ります．このファイルを「**ソース・ファイル**」と呼び，普通は拡張子を「**.ASM**」にします．

テキスト・ファイル
Text File

このファイルはテキスト・ファイル（形式）といって，文字だけで構成されているので，我々が普通に読めるファイルです．

オブジェクト・ファイル

オブジェクト・ファイル
Object File

この「ソース・ファイル」から拡張子が「**.OBJ**」のファイルを生成します．

この作業をするプログラムを**アセンブラ**といいます．この工程を「**アセンブル**」，「**アセンブルする**」などと呼び，生成された拡張子「**.OBJ**」のファイルを「**オブジェクト・ファイル**」と呼んでいます．

アセンブラ
Assembler

バイナリ・ファイル
Binary File

オブジェクト・ファイルはバイナリ・ファイル（文字以外のコードで構成されたファイル）なのでエディタなどで普通に読むことはできません．

オブジェクト・ファイルはソース・ファイルにエラーがあるときは生成されません．そのときはエディタでソース・ファイルを修正し，再び「アセンブル」します．これをアセンブル・エラーがなくなるまで繰り返さなければなりません．

リンク

リンク
Link

複数のオブジェクト・ファイルを結合し，実行形式のファイルを作ります．

リンカ
Linker

この工程を「**リンク**」といい，これを行なうプログラムを「**リンカ**」といいます．

アセンブラ・プログラムのパッケージには，普通アセンブラ以外に「ライブラリアン」，「リンカ」が含まれています（例：**tasm**のパッケージには**tlib**（ライブラリアン）と**tlink**（リンカ）が含まれています）．

ライブラリアン
Librarian

ソース・ファイルが1つであっても，必ず，

ソース・ファイル → オブジェクト・ファイル → 実行ファイル

という工程が必要です．

プログラムは複数のソース・ファイルに分割して書かれることが多く，ソース・ファイルの数だけオブジェクト・ファイルが生成されます．それらのオブジェクト・ファイルを結合しアドレスの調整（**JMP ???, MOV [???],AX**など???の値を正しくセットする）をするのがリンクです．

複数のソース・ファイルによる構成の開発では通常メイク・ファイルが利用されます．

メイク・ファイル
Make File
ソース・ファイルなどの依存関係を記述し，プログラム開発を管理するためのファイル．

＊参照　メイク・ファイル

5　システム（パソコン）の設定

パソコンを使い始めた初心者が最初に困るのが，「**CONFIG.SYS**」と「**AUTOEXEC.BAT**」です．

やっていることはそれほど難しいことではないのですが，勝手に「**CONFIG.SYS**」や「**AUTOEXEC.BAT**」を書き換える“行儀の悪い”ソフトが多く，混乱に拍車をかけている気がします．

CONFIG.SYS
「コンフィグ・シス」と読むことが多い．

AUTOEXEC.BAT
「オートエグゼック・バット」と読むことが多い．

「**CONFIG.SYS**」では基本的な環境（バッファの確保など）の設定，デバイスドライバの組み込みによる機能の拡張などをします．

MS-DOS Ver.5でCPUが80386(486)の場合

FILES=30	同時にオープンできるファイル（エリアの確保）
BUFFERS=20	バッファの確保
BREAK=OFF	CTRL-Cブレイクの無効
DOS=HIGH,UMB	＊
DEVICE=A:¥DOS¥HIMEM.SYS	＊
DEVICE=A:¥DOS¥EMM386.EXE /M=4848 /UMB	＊
SHELL=A:¥COMMAND.COM A:¥ /E:512 /P	コマンド・プロセッサの設定
DEVICE=A:¥DOS¥RAMDISK.SYS 2048 512 /E	
DEVICE=A:¥DOS¥SMARTDRV.SYS 2048 512 /E	＊
DEVICEHIGH=A:¥DOS¥PRINT.SYS	プリンタが使える
DEVICEHIGH=A:¥DOS¥RSDRV.SYS	RS-232C（通信）が使える
DEVICEHIGH=A:¥DOS¥MOUSE.SYS	マウスが使える
LASTDRIVE=E	仮想ドライブ最大値の設定

＊印はMS-DOS Ver 5特有

MS-DOS Ver.4以前 またはCPUが80386(486)でない場合

FILES=30	同時にオープンできるファイル（エリアの確保）
BUFFERS=20	バッファの確保
BREAK=OFF	CTRL-Cブレイクの無効
SHELL=A:¥COMMAND.COM A:¥ /E:512 /P	コマンド・プロセッサの設定
DEVICE=A:¥DOS¥PRINT.SYS	プリンタが使える
DEVICE=A:¥DOS¥RSDRV.SYS	RS-232C（通信）が使える
DEVICE=A:¥DOS¥MOUSE.SYS	マウスが使える
LASTDRIVE=E	仮想ドライブ最大値の設定

CONFIG.SYSの詳細

FILES
「ファイルズ」と読む

FILES=30

同時にオープンできるファイル（ハンドル）の最大数を決めます．

この数が少ないと，同時にたくさんのファイルを使うアプリケーション・プログラムでは正常に動作しないことがあります．

通常は，「**FILES=30**」で問題ないと思います．もし動作しないアプリケーションがあれば多くしてください．

BUFFERS
「バッファーズ」と読む

BUFFERS=20

DOSがファイル・アクセスなどに使用するバッファ領域の大きさです．

この効果を確認するには，**COPY**コマンドで大きなファイルをフロッピーディスクにコピーしてみてください．

90Kバイトのファイルであれば「20K×4回＋10K」で計5回フロッピーに“ガチャガチャ”と書き込むはずです．

メモリ・マネージャ
Memory Manager

ハイメモリ・エリア
High Memory Area

CPUが80386以上のマシンでMS-DOS Ver.5やメモリ・マネージャを使用すると，このバッファをハイメモリ・エリアに置くこともできます．この場合のオプションの指定などは使うメモリ・マネージャなどで違うので，それらのマニュアルを参照してください．

BREAK=OFF

わけの分からないトラブルの1つの解決法がこれだと言われています．

CTRL-Cは通常，プログラムの中止（強制終了）に使用されます．

エディタなど多くのソフトではCTRL-Cは画面操作のコントロールなどに使われています．

タイミングの問題なのかどうなのか，CTRL-Cの扱いはけっこう不透明な部分があって，因果関係は分かっていません．

ちょうど，昔の人がちょっと体の具合が悪いと「正露丸」を飲むようなものです．軽く考えておいてください．

DOS=HIGH,UMB

MS-DOS Ver.5以降でCPUが80286以上の機種でのみ使えます．

“**HIGH**”はMS-DOSのシステム・プログラムをハイメモリ・エリアに置く指定です．いわゆる“コンベンショナル・メモリ”がその分広くなります．

“**UMB**”はUMB（Upper Memory Block）を使う指定で，これを指定するとUMBにデバイス・ドライバを置けるようになり，“コンベンショナル・メモリ”を広くできます．この機能はCPUが80386以上でのみ使えます．

また，この指定をするときは「**EMM386.EXE**」に「/UMB」を指定しなければいけません（PC-9801の場合）．

コンベンショナル・メモリ
Conventional Memory

DEVICE=A:¥DOS¥HIMEM.SYS

MS-DOS Ver.5以降でCPUが80286以上の機種でのみ使えます．

XMSの使用が可能になります．

XMS
eXtended Memory Specification の略
80286以降のCPUで10000h以上のアドレスのメモリをアクセスするための規格．
HMAとEMBがある．

DEVICE=A:¥DOS¥EMM386.EXE /M=4848 /UMB

「**EMM386.EXE**」はプロテクト・メモリをEMSに割り当てるプログラムです．この指定をするときはこの行より前に**HIMEM.SYS**を記述しておかなければいけません．

Protected Mode Memory
プロテクト・メモリ（和製英語，英語ではExtended Memory）

1番目のパラメータの「**/M=4848**」はEMSに割り当てるプロテクト・メモリのサイズ指定で，単位はKバイトです．

2番目のパラメータの「**/UMB**」はUMBを使うための指定です．

EMS
Expanded Memory Specification の略
MS-DOSの管理する1MBの領域内で拡張メモリを扱うための規格．

SHELL=A:¥COMMAND.COM A:\ /E:512 /P

「**COMMAND.COM**」はキーボードから入力された文字列の解析（解釈）や，それをMS-DOSに伝える処理をするプログラム（コマンド・インタープリタ）で，「常駐部」と「非常駐部」からなっています．

UMB
Upper Memory Block の略
MS-DOSの管理できる1MBのメモリ領域のうちシステム予約領域中の空き領域にメモリを割り当てたもの．

1番目のパラメータの「**A:¥**」は非常駐部が必要なときに「**COMMAND.COM**」をディスクのどのパス（ドライブ：ディレクトリ）から読むかを指定しています．

2番目のパラメータの「**/E:512**」は「**SET=**」や「**PATH=**」でセットされる環境変数のエリアの大きさを指定しています．

「環境変数のエリアが足りない」のメッセージが出たり，SETコマンドで確認して，設定しているはずの環境変数が表示されないときは，環境変数のエリアが小さいと判断してください．

そのときはこの“512”という数字を大きくしてください．

3番目の「/P」は「**COMMAND.COM**」が常駐するという意味ですが，詳細は分かりません．

*

とにかく，このスタイルが一般的だと考えてください．必要に応じてパスの部分と数字を書き換えてください．

パス
PATH

DEVICE=A:¥DOS¥RAMDISK.SYS 2048 512 /E

「**RAMDISK.SYS**」はプロテクト・メモリまたはEMSメモリをRAMディスクとして使うデバイスドライバです．

1番目のパラメータの「2048」はRAMディスクのサイズで，単位はKバイトです．この場合は，2048Kバイトです．

2番目のパラメータの「512」はRAMディスクのセクタのサイズです．この場合は，1セクタのサイズは512バイトとなります．

3番目のパラメータの「/E」はEMSメモリからRAMディスクに割り当てるという意味です．

DEVICE=A:¥DOS¥SMARTDRV.SYS 2048 512 /E

「**SMARTDRV.SYS**」はディスク・キャッシュのプログラムです．

1番目のパラメータの「**2048**」はキャッシュの標準サイズで，単位はKバイトです．この場合は，2048Kバイトです．

2番目のパラメータの「**512**」はキャッシュの最小サイズです．この場合は，512Kバイトになります．

3番目のパラメータの「**/E**」はEMSメモリからディスク・キャッシュに割り当てるという意味です．

DEVICEHIGH=A:¥DOS¥PRINT.SYS

「**PRINT.SYS**」を組み込むとプリンタが使えるようになります（PC-9801）．

MS-DOS Ver.5未満またはVer.5でもCPUが**80286**以下のものでは，「**DEVICEHIGH**」の代わりに「**DEVICE**」を記述します．

DEVICEHIGH=A:¥DOS¥RSDRV.SYS

「**RSDRV.SYS**」を組み込むとRS-232C（通信）が使えるようになります（PC-9801）．

DEVICEHIGH=A:¥DOS¥MOUSE.SYS

マウスを使うときは，この記述が必要です．

LASTDRIVE=E

MS-DOSがアクセスする最後のドライブ名を指定します．ただし接続されているドライブの数より小さい値は指定できません．

この場合は，「F:」以降のドライブ名が使えなくなります．

AUTOEXEC.BAT の詳細

```
SET COMSPEC=A:¥COMMAND.COM
```

「**COMSPEC**」は「**COMMAND.COM**」を読み込むときの指定です．

「**COMMAND.COM**またはそれに相当するものをを読み込むときは**COMSPEC=**で定義されたものを読みましょう」という決まりです．

普通は「**CONFIG.SYS**」の「**SHELL=**」で指定したファイルが自動的に設定されますが，RAMディスクなどに「**COMMAND.COM**」を置いて，そちらを指定すれば読み込みが速くなります．

```
LOADHIGH A:¥DOS¥DOSKEY.COM
```

「**DOSKEY.COM**」はMS-DOS Ver.5で追加された機能です．

このプログラムはメモリに常駐して，以前に入力したコマンドを記憶しておいてくれます．コマンド入力が少し楽になりますが，それほど便利なものではない，というのが実感です．

```
SET MIFES=A:¥MIFES
SET MIMACRO=A:¥MIFES
SET TMP=H:¥
SET MAN=A:¥MAN
SET PAGER=CMORE
SET CCT=A:¥CCT
SET TEMP=A:¥DOS
SET DOSDIR=A:¥DOS
```

このあたりは各アプリケーション向けの環境変数の設定です．

各環境変数の意味はアプリケーションのマニュアルを見ないと分かりません．

```
PATH=E:¥;A:¥BC¥BIN;A:¥;A:¥DOS;A:¥TOOLS;A:¥MTOOL;A:¥MIFES;A:
¥ZASM3;A:¥CCT;
```

コマンド・パスの設定です．1つ1つのパスは「;」（セミコロン）で区切って記述します．

プロンプトから，

```
A>ZZ386
```

とキーをタイプすると，最初にカレント・ディレクトリの「**ZZ386.EXE**」を探し，次に，

```
E:¥ZZ386.EXE A:¥BC¥BIN¥ZZ386.EXE A:¥ZZ386.EXE .....
```

とファイルを探します．見つかった時点で，そのファイルを読み込み実行します．

最後まで見つからなかったときは，

```
コマンドまたはファイル名が違います.
```

というメッセージを出力します．

注：実際は「**.COM**」「**.BAT**」も探しています．

```
PROMPT $e[15C$e[21m$p$e[22m : $d: $t$h$h$h$e[m$_$n$g
```

DOSのプロンプトを変更しています．

カレント・ディレクトリ，日付をいつも表示しておいた方が，トラブルを避けられるし便利なのでこうしています．

エスケープ・シーケンス
Escape Sequence

「**$e[15....**」の部分はエスケープ・シーケンスで記述してあります．

これはPC-9801の例ですが，使用機種によってかなり違います．IBM系（AX，DOS/V）では**CONFIG.SYS**に「**ANSI.SYS**」を組み込む必要があります．

詳しくは，各機種のMS-DOSのマニュアルを見てください．

IN JCXZ
AHF LDS LEA
CK LODS LOOP
NOP NOT OUT
SH PUSHF REP
EG STOS WAIT

第2部　まずニモニックを少しだけ覚えましょう

8086のニモニックはざっと300個はあります．これをいきなり覚えるのは嫌になります．私などはこれだけでも投げ出したくなりました．そのうえ，覚えたからといってすぐに使えるわけでもありません．さらにアドレッシング・モードがあります．おっと，いきなり“アドレッシング・モード”などと面倒な言葉を使ってしまいました．とりあえず，「変形」，「何とかの5段活用」ぐらいに考えておいてください．

XCHG XLAT
ADD CMP CMPS
SBB SCAS SUB
OR SAR SHL
TEST XOR
INC

1 機能から見たニモニック(命令)の分類

ニモニック
Mnemonic

ニモニックとは，CPUの命令を少しだけ人間に理解しやすく表わしたものです．ニモニックはCPUの命令（最小単位）に1対1で対応しています．1つ1つの命令を見ても何の意味ももっていいません．何個かの命令が組み合わさらないと意味のある結果とならないのです．

平たく言えば，アセンブリ言語には画面に「こんにちは」と表示する命令はないのです．

BASICでは，

```
PRINT "こんにちは"
```

C言語では，

```
printf("こんにちは");
```

ですみますが，アセンブラでは，

```
MOV       DX,MSG
MOV       AH,09H
INT       21H

MSG  DB   "こんにちは",'$'
```

これだけ書かなければなりません．

アセンブラでプログラムを書くにはそれだけでは何に使えるのか分からないような，それでいて何にでも使える貧弱な命令（ニモニック）を組み合わせて，希望する機能を実現します．

昔使った『豆単』で英単語を覚えるようにニモニックを覚える方法もありますが，ちょっと辛いものがあるので，機能別に分類してみましょう．

①データのコピー
②算術演算命令
③論理演算命令
④ストリング命令
⑤ジャンプ，コール命令
⑥プロセッサ制御命令
⑦入出力命令
⑧割り込み命令
⑨シフト命令

このようにグループとして考えてみましょう．

2 データのコピー命令

MOVなどデータをコピー（転送）する命令は最も多く使われる命令グループです．

CRT画面に文字を書いたり，データを一時的に記憶したり，取り出したりします．

データのコピーは，ソース（コピー元），デスティネーション（コピー先）の種類によって次の5つが考えられます．

ソース
source

ディスティネーション
destination

ソース		デスティネーション
①レジスタ	→	レジスタ
②レジスタ	→	メモリ
③メモリ	→	レジスタ
④数値	→	レジスタ
⑤数値	→	メモリ

その他に8086ではセグメント・レジスタを考慮する必要があります．

セグメント・レジスタは特殊なレジスタなので，直接数値をコピーすることはできません．

したがって，コピー元（ソース）には，「汎用レジスタ，メモリの内容，数値，セグメント・レジスタ」の4つがあります．

コピー先（ディスティネーション）には，「汎用レジスタ，メモリ，セグメント・レジスタ」の3つがあります．

汎用レジスタ
General Register

MOV命令はこれらを組み合わせて使います．

ただし，中には使えない組み合わせがあります．次の図の×が付いているところが使えない組み合わせです．

「CPUの直行性」と言われることがありますが，×印の付いているところが直行性の悪いところです．

MOV命令は×印の組み合わせだけを頭に入れておいて，コピー元，コピー先を自由に組み合わせて軽く使いましょう．

		コピー先		
		汎用レジスタ	メモリ	セグメント・レジスタ
コピー元	汎用レジスタ	○	○	○
	メモリ	○	×	○
	セグメント・レジスタ	○	○	×
	数値	○	○	×

参照　汎用レジスタ

メモリの指定方法＝アドレッシング・モード

アドレス
Address

アドレッシング・モード
Addressing Mode

メモリの特定の場所を「**アドレス**」と言い，そのアドレスの指定方法を「**アドレッシング・モード**」と言います．

コピー元またはコピー先にメモリを使うとき，このアドレッシング・モードを知る必要があります．

アドレッシング・モードは大きく「**直接アドレッシング**」と「**レジスタ間接アドレッシング**」のふたつに分けられます．

直接アドレッシング

直接アドレッシングはその呼び方のとおり直接アドレスを指定します．
実際にはラベル名で指定します．

```
MOV     [ラベル名],AX
MOV     AX,[ラベル名]
```

レジスタ間接アドレッシング

レジスタ間接アドレッシングは，レジスタのもつ値をアドレスの値として指定します．

アドレスの値として使うレジスタは，**BX**レジスタ，ポインタ・レジスタ（**BP**），インデックスレジスタ（**SI,DI**）を組み合わせて使え，それに8または16ビットのディスプレースメント（数値）を付けることができます．

汎用レジスタ
General Register

レジスタ間接のレジスタ組み合わせ

	BX	BP	SI	DI
BX	−	×	○	○
BP	×	−	○	○
SI	○	○	−	×
DI	○	○	×	−

上図のように，使える4つのレジスタはインデックス・レジスタとそれ以外に区別できます．インデックス・レジスタ同士（**SI,DI**）や，それ以外のレジスタ同士（**BX,BP**）の組み合わせは使えません．

```
MOV     AX,[組み合わせ]
MOV     AX,[組み合わせ＋符号付き1バイト]
MOV     AX,[組み合わせ＋符号なし2バイト]
```

セグメント・プリフィックス

セグメント・プリフィックス
Segment Prefix

8086が一度にアクセスできるメモリは，セグメント・レジスタごとにそれぞれ64Kバイトです．これらは暗黙の了解で，データは**DS**レジスタの指すデータ空間，ジャンプなどの命令は**CS**レジスタの指すメモリ空間を対象にします．

8086は**CS,DS,SS,ES**の4つのセグメント・レジスタがあるので，同時に4つの空間を持つことができます．これらは重なる（同じ空間を指す）ことがあります．「comモデル」の実行ファイルでは通常4つのセグメント・レジスタは同じ値になります．

ここで述べているのはデータのコピーなので，セグメント間ジャンプなどについては後述します．

データは**DS**の指す「データ・セグメント」内が対象となるので，それ以外の**ES,SS,CS**の空間を使うときはあらかじめ指定する必要があります．これが「**セグメント・プリフィックス**」です．

```
MOV     AX,CS:[ラベル名]
MOV     ES:[BP+SI],CX
MOV     AX,SS:[BX]
```

などです．ここで，「**CS:**」，「**ES:**」，「**SS:**」など「＿」の部分がプリフィックスです．

データ・セグメントを使うときに「**DS:**」のプリフィックスを記述して分かりやすくしてもかまいません．もしそれが不要であればちゃんとアセンブラが無視してくれますから．

3 算術演算命令

これらの命令には，

①加算
②減算
③乗算
④除算
⑤比較

の5グループの命令があります．

加算命令

加算の命令グループにはつぎのものがあります．

ADD
ADC
INC
AAA
DAA

ADC
ADd with Carry

ADD,ADC

形式は

ADD	**DEST, SOC**
ADC	**DEST, SOC**

で，**ADD**は，

ソース（SOC）＋デスティネーション（DEST）→デスティネーション

という演算をし，**ADC**は，

ソース＋デスティネーション＋キャリーフラグ→デスティネーション

という演算をします．

2つの違いはキャリー・フラグも同時に加算されるかどうかの違いです．

キャリー・フラグは「1」か「0」のいずれかです．

オーバーフロー
Overflow

いずれもオーバーフロー（桁上がり）があるとキャリー・フラグがセット（1）され，それ以外ではリセット（0）されます．

ソースとディスティネーションの組み合わせとしては，

ADD(ADC)	メモリ/レジスタ	，レジスタ
ADD(ADC)	レジスタ	，メモリ/レジスタ
ADD(ADC)	メモリ/レジスタ	，数値

ができます．メモリ同士の演算はできません．

INC

形式は，

INC	メモリ/レジスタ

で，メモリまたはレジスタを「1」加算します．

オペランドとして使えるものは，

INC	汎用レジスタ
INC	8ビットレジスタ
INC	レジスタ間接アドレッシング

のいずれかです．

AAA(Ascii Adjust for Addition)
DAA(Decimal Adjust for Addition)

これらはともに加算後の結果を調整する命令です．

AAAは**AX**レジスタを対象に「アンパックBCD」加算した結果を調整します．
DAAは**AL**レジスタを対象に「パックドBCD」加算した結果を調整します．

どちらの命令も以下の例のようにADDまたはADC（加算命令）を実行した直後にだけ使います．

AAAの場合		DAAの場合	
ADC	AX,BX	ADC	AL,BL
AAA		DAA	
または		または	
ADD	AX,BX	ADD	AL,BL
AAA		DAA	

CPUの計算は16進（2進）で行なわれていますが，**AAA**または**DAA**を使うと10進で行なわれたように結果を調整します．**AAA**の場合，8ビットを1桁として，**DAA**の場合4ビットを1桁として調整します．

［例］「**AX=0108h,BX=0508h**」としてAAA命令を実行した結果

AAAを実行しない	**AAA**を実行する
0108(h)	**01 08 (=18)**
+ 0508(h)	**+ 05 08 (=58)**
0610(h)	**07 06 (=76)**

［例］「**AL=18h,BL=58h**」としてDAA命令を実行した結果

DAAを実行しない	**DAA**を実行する
18(h)	**18**
+ 58(h)	**+ 58**
70(h)	**76**

DAAはよく使いますが，**AAA**について私は有効な使用例を知りません．

減算命令

SUB,SBB

SUB
SUBtract

SBB
SuBtract with Borrow

形式は，

```
SUB  DEST,SOC
SBB  DEST,SOC
```

で，**SUB**は，

〔デスティネーション(DEST)〕－〔ソース(SOC)〕→デスティネーション

という演算をし，SBBは，

デスティネーション － ソース － キャリー・フラグ→デスティネーション

という演算をします．

2つの違いはキャリー・フラグも同時に減算されるかどうかの違いです．

演算結果がマイナスになった場合はキャリー・フラグが，演算結果が「0」の場合はゼロフラグが1になります．

ソースとディスティネーションの組み合わせとしては，

SUB(SBB)	メモリ/レジスタ	，	レジスタ
SUB(SBB)	レジスタ	，	メモリ/レジスタ
SUB(SBB)	メモリ/レジスタ	，	数値

ができます．メモリ同士の演算はできません．

DEC
DECrement

DEC

形式は，

DEC	メモリ/レジスタ

で，メモリまたはレジスタを「1」減算します．結果がゼロ・フラグに反映されます（結果が「0」ならゼロ・フラグが「1」になる）．

乗算命令

乗算の命令には，

MUL
IMUL
AAM

がありますが，あまり使わないので省略します．

MUL
unsigned MULtiplication

IMUL
Integer MULtiplication

AAM
Ascii Adjust for Multiply

除算命令

除算の命令には

DIV
IDIV
CBW
CWD
AAD

がありますが，あまり使わないので省略します．

DIV
DIVision

IDIV
Integer DIVision

CBW
Convert Byte to Word

CWD
Convert Word to Double word

AAD
Ascii Adjust for Division

比較命令

形式は，

CMP	比較される値	，	比較する値

で，**SUB**命令と同じ演算をしますが，演算の結果はセットされず，フラグの値のみ変化します．

CMP
CoMPare

4 論理演算命令

論理演算命令には次のものがあり，ビットごとのAND，OR，XOR演算をします．

AND
NOT
OR
XOR
TEST

AND（論理積命令）

形式は，

AND	データ1，データ2

で，結果は，

データ1＝　データ1 ∧ データ2

となります．

「データ1」と「データ2」の各ビットごとの論理積（AND）をとり，結果を「データ1」にセットします．

AF以外のすべてのフラグに影響を与えます．

AND命令には3つのタイプがあり，それぞれバイトまたはワードの論理演算をします．

AND	メモリ/レジスタ	，	レジスタ
AND	レジスタ	，	メモリ/レジスタ
AND	メモリ/レジスタ	，	数値

NOT（否定命令）

形式は，

NOT	データ

で，結果は，

データ＝$\overline{\text{データ}}$

となります．

データにはデータの1の補数（すべてのビットを反転した値）がセットされます．

フラグは変化しません．

NOT命令は1つのタイプだけです．バイトまたはワードの演算ができます．

NOT	メモリ/レジスタ

OR（論理和命令）

形式は，

OR	データ1，データ2

で，結果は

データ1＝データ1 ∨ データ2

となります．

「データ1」と「データ2」の各ビットごとの論理和（OR）をとり，結果を「データ1」にセットします．

どちらかのビットが「1」であれば「1」に，どちらのビットも「0」であれば「0」になります．

AF以外のすべてのフラグに影響を与えます．

OR命令には3つのタイプがあり，それぞれバイトまたはワードの論理演算をします．

OR	メモリ/レジスタ	，	レジスタ
OR	レジスタ	，	メモリ/レジスタ
OR	メモリ/レジスタ	，	数値

XOR
logical eXclusive OR

XOR（排他的論理和命令）

形式は，

XOR	データ1，データ2

で，結果は，

データ1＝　データ1 ∀ データ2

となります．

「データ1」と「データ2」の各ビットごとの排他的論理和（XOR）をとり，結果を「データ1」にセットします．

両方のビットが同じ（ともに「1」か，ともに「0」）であれば「0」に，違っていれば「1」にセットします．

AF以外のすべてのフラグに影響を与えます．

XOR命令には3つのタイプがあり，それぞれバイトまたはワードの論理演算をします．

XOR	メモリ/レジスタ	，	レジスタ
XOR	レジスタ	，	メモリ/レジスタ
XOR	メモリ/レジスタ	，	数値

TEST

形式は，

TEST	データ1，データ2

で，**AND**命令と似ていますが，「データ1∧データ2」の結果は「データ1」にはセットされません．つまり，「データ1」は変更されず，フラグだけが変更されます．

AF以外のすべてのフラグに影響を与えます．

TEST命令には3つのタイプがあり，それぞれバイトまたはワードの論理演算をします．

TEST	メモリ/レジスタ	，	レジスタ
TEST	レジスタ	，	メモリ/レジスタ
TEST	メモリ/レジスタ	，	数値

5 ストリング命令

ストリング命令には，

ストリング・プリミティブ命令
リピート・プリフィックス命令

の2種類があります．

ストリング・プリミティブ命令

ストリング・プリミティブ命令は，暗黙の了解で使われるレジスタ（SI，DI）をデータへのポインタとして使い，コピーや比較などを行ない，実行サイズ（バイト，ワード）に合わせてポインタの増減をする複合的な命令です．

ストリング・プリミティブ命令には次の種類があり，各命令（ニモニック）の末尾に「**B**」または「**W**」を付けることで，その命令の単位（バイト，ワード）を指定します．

これらの命令は後述するリピート・プリフィックス命令と組み合わせ，連続したデータのコピー，比較などを効率良く行なうのに使われます．

MOVS	メモリ	←	メモリのコピー
LODS	**AX(AL)**	←	メモリのコピー
STOS	メモリ	←	**AX(AL)**のコピー
SCAS	**AX(AL)**	←→	メモリの比較
CMPS	メモリ	←	メモリの比較

MOVS
MOVe String

MOVS

MOVSは，バイトまたはワードのデータを「**DS:SI**」で示されるアドレスから「**ES:DI**」で示されるアドレスにコピーします．

ES:[DI] ← DS:[SI]

「ディレクション・フラグ=0」のとき

DI=DI+1 または**+2**
SI=SI+1 または**+2**

「ディレクション・フラグ=1」のとき

DI = DI-1または **-2**
SI = SI-1 または **-2**

MOVSB
MOVe String Byte

MOVSW
MOVe String Word

MOVSB	バイト・データのコピー
MOVSW	ワード・データのコピー

LODS
LOaD String

LODS

LODSはバイトまたはワードのデータを「**DS:SI**」で示されるアドレスから**AL**または**AX**レジスタにコピーします．

AL ← DS:[SI]

または

AX ← DS:[SI]

「ディレクション・フラグ=0」のとき

SI=SI+1または**+2**

「ディレクション・フラグ=1」のとき

SI=SI-1または**-2**

LODSB
LOaD String Byte

LODSW
LOaD String Word

LODSB	バイト・データのコピー
LODSW	ワード・データのコピー

STOS
STOre String

STOS

STOSは**AL**（バイト）または**AX**（ワード）のデータを「**ES:DI**」で示されるアドレスへコピーします．

ES:[DI]←ALまたは**AX**

「ディレクション・フラグ=0」のとき

DI=DI+1または**+2**

「ディレクション・フラグ=1」のとき

DI=DI-1または**-2**

STOSB
STOre String Byte

STOSW
STOre String Word

STOSB	バイト・データのコピー
STOSW	ワード・データのコピー

SCAS

SCAS
SCAn String

SCASは**AL**（バイト）または**AX**（ワード）の内容と「**ES:DI**」で示されるアドレスのデータとを比較します．

すべての算術演算フラグに影響を与えます（CMP命令と同様）．

ES:[DI]とAL（または**AX**）を比較

「ディレクション・フラグ=0」のとき

DI=DI+1または**+2**

「ディレクション・フラグ=1」のとき

DI=DI-1または**-2**

SCASB	バイト・データの比較
SCASW	ワード・データの比較

SCASB
SCAn String Byte

SCASW
SCAn String Word

CMPS

CMPS
CoMPare String

CMPSはバイトまたはワードのデータを「**DS:SI**」で示されるアドレスの内容と「**ES:DI**」で示されるアドレスの内容を比較します．

ES:[DI]とDS:[SI]を比較

「ディレクション・フラグ=0」のとき

DI=DI+1または**+2** **SI=SI+1**または**+2**

「ディレクション・フラグ=1」のとき

DI=DI-1または**-2** **SI=SI-1**または**-2**

CMPSB	バイト・データの比較
CPMSW	ワード・データの比較

CMPSB
CoMPare String Byte

CMPSW
CoMPare String Word

比較命令の方向

比較命令で戸惑うのは大小の方向性です．具体的には，「どちらが大きいときにキャリーフラグが1になるか」です．

CMP　　データ１，データ２

のときは比較的分かりやすく，「データ１＜データ２」のとき，キャリーフラグは「1」になります．

プログラムを書いているときも，ときどき間違うのが，**SCAS**と**CMPS**です．

SCASは「**AL,ES:[DI]**」ですから，「**AL<ES:[DI]**」のとき，キャリー・フラグは「1」になります．

CMPSは「**DS:[SI],ES:[DI]**」ですから，「**DS:[SI]<ES:[DI]**」のとき，キャリー・フラグは「1」になります．

CLDとSTD

一連のストリング・プリミティブ命令を実行するときデータへのポインタとして使われる**SI**，**DI**レジスタは，ディレクション・フラグの値によって自動的に増減されます．

ディレクション・フラグをクリア（0）やセット（1）するのがこの命令です．

CLD
CLear Direction flag

STD
Set Direction flag

CLD	ディレクション・フラグ=0，**SI**，**DI**を＋する．
STD	ディレクション・フラグ=1，**SI**，**DI**を－する．

リピート・プリフィックス命令

リピート・プリフィックス命令には次の3つがあり，条件を満たす間，この命令に続くストリング・プリミティブ命令の実行を続けます．

REP
REPeat

REPE
REPeat string operation while Equal

REPZ
REPeat string operation while Zero

REPNE
REPeat string operation while Not Equal

REPNZ
REPeat string operation while Not Zero

REP
REPE REPZ（同義）
REPNE REPNZ（同義）

MOVS，**LODS**，**STOS**のコピー（ストリング・プリミティブ命令）には**REP**を使い，**SCAS**，**CMPS**の比較（ストリング・プリミティブ命令）には**REPZ**，**REPNZ**を使います．

REP

REPは**CX**レジスタをカウンタとして使います．ストリング・プリミティブ命令を1回実行するたびに**CX**レジスタは「-1」されます．

CXレジスタが0になると，ストリング・プリミティブ命令の実行は中止されます．

［MOVSの例］

データSからデータDに20バイトのデータをコピーする．

```
CLD                             ;ディレクション・フラグを増加方向に
MOV     SI, OFFSET DATA_S       ;DS:SI=データS
MOV     AX, SEGMENT DATA_D      ;コピー先のセグメント
MOV     ES, AX
MOV     DI, OFFSET DATA_D       ;コピー先のオフセット
MOV     CX, 20                  ;コピーするバイト数
REP     MOVSB                   ;コピーの実行
```

必要なら**DS**の値も設定します．

REPE
REPZ（同義）

CXレジスタをカウンタとして使うのに加え，比較結果のゼロ・フラグが使われます．

比較のストリング・プリミティブ命令を1回実行するごとに**CX**レジスタは「-1」されます．

「CX≠0」かつ「比較結果が等しい」とき

比較のストリング・プリミティブ命令の実行を続けます．

「CXレジスタが0になる」か「ゼロフラグが0になる（等しくない）」とき

ストリング・プリミティブ命令の実行は中止されます．

具体的には2つのデータ列（文字列など）を比較し，「指定された長さ比較が終わる」（等しかった）か，「等しくないデータに出会う」まで，比較のストリング・プリミティブ命令 が繰り返されます．

このプリフィックスで**CX**レジスタは「-1」されますが，例外的にフラグ（ゼロ・フラグ）には影響を与えません．ゼロ・フラグは比較の結果のみを反映しています．

［CMPSの例］

20バイトのデータ1とデータ2が等しいかどうかを調べる．

```
        CLD                             ;ディレクション・フラグを
                                        ;増加方向
        MOV     SI, OFFSET DATA_1       ;ＤＳ：ＳＩ＝データ１
                                        ;必要なＤＳ の値も設定
        MOV     AX, SEGMENT DATA_2      ;データ２のセグメント
        MOV     ES, AX
        MOV     DI, OFFSET DATA_2       ;データ２のオフセット
        MOV     CX, 20                  ;比較するバイト数
        REPZ    CMPSB                   ;比較の実行
        JZ      EQUAL
        ...     等しくない結果の処理
EQUAL:
        ...     等しい結果の処理
```

REPNE
REPNZ（同義）

CXレジスタをカウンタとして使うのに加え，比較結果のゼロ・フラグが使われます．

比較のストリング・プリミティブ命令を1回実行するたびに，**CX**レジスタは「-1」されます．

「CX≠0」かつ「ゼロフラグが0（比較結果が等しくない）」とき

比較のストリング・プリミティブ命令の実行を続けます．

「CXレジスタが0になる」か「ゼロフラグが1になる（等しい）」とき

ストリング・プリミティブ命令の実行は中止されます．

具体的にはデータ列（文字列など）から 特定のデータ（文字など）を探すときなどに使われます．

データを検索し，指定された長さ検索が終わるか（見つからなかった），データが見つかるまで比較のストリング・プリミティブ命令が繰り返されます．

［SCASの例］

20バイトの文字列から「A」という文字を探す．

```
        CLD                         ;ディレクション・
                                    ;フラグを増加方向
        MOV     AX,SEGMENT  STRING_S        ;文字列のセグメント
        MOV     ES, AX
        MOV     DI, OFFSET  TRING_S         ;文字列のオフセット
        MOV     CX, 20              ;検索するバイト数
        MOV     AL, 'A'             ;A を検索する
        REPNZ   SCASB               ;検索の実行
        JZ      FOUND
        ...     見つからなかった処理
FOUND:
        ...     見つかった処理，アドレスはES:DIに保持されている
```

6　ジャンプ，コール命令

ジャンプ関連の命令には次のものがあります．

CALL
RET
JMP
条件ジャンプ

CALL命令

CALL命令は現在実行中のプログラムのアドレス（**CALL**命令の次の命令のアドレス（「**IP:CS**」または「**IP**」））を自動的にスタックに記憶します．

CALLには「セグメント間**CALL**」と「セグメント内**CALL**」の2種類があります．いわゆる「FAR CALL」，「NEAR CALL」といわれているものです．

「FAR CALL」はその名のとおり「遠くを呼ぶ」わけで，他のセグメント・グループにあるサブルーチンを呼び出すのに使用されます．

COMモデルやスモール・モデルのプログラムでは，あまり使うことはありません．

プログラムの記述では特に断わらなくても，アセンブラが正しく処理をしてくれます．

```
CALL    プロシージャ名　または　ラベル名
```

将来的（大きなプログラムを書くとか）にはFAR，NEARを明記しておいた方が良いと思います．

```
CALL    NEAR PTR プロシージャ名　または　ラベル名
CALL    FAR PTR プロシージャ名　または　ラベル名
```

RET命令

RET命令は**CALL**命令に対応し，「**FAR**」（セグメント間），「**NEAR**」（セグメント内）があります．

RET
RETurn

FARでは**CALL**命令でスタックに記録されていた（**CS:IP**）が復元（**POP**）され，サブルーチンを呼び出したプログラムに制御が戻ります．

NEARでは（**IP**）だけが復元されます．

```
RETF
RETN
```

と距離を明示してもかまいませんが，普通はしません．

呼び出される側の**PROC**で表現します（参照：第3部～）．

```
プロシージャ名          PROC   NEAR(またはFAR)
   ... プログラム本体

   RET
プロシージャ名          ENDP
```

JMP
JuMP

JMP命令

JMP命令を距離で分類すると3つになります.

FAR	セグメント間ジャンプ すべての空間にジャンプ可能
NEAR	セグメント内ジャンプ，2バイトのディスプレースメント 現在のセグメント内，**CS:0000～CS:FFFF**の範囲
SHORT	セグメント内ジャンプ，1バイトのディスプレースメント **CS:IP-128～CS:IP+127**の範囲

ジャンプ先の指定方法で分類すると3つになります.

イミディエイト・ジャンプ

直接，ジャンプ先のアドレスを指定します.

実際のプログラミングではラベル名，プロシージャ名が使われます.

```
JMP     ラベル名
JMP     プロシージャ名
```

このジャンプではすべての距離のジャンプが使えます.

```
JMP     FAR PTR     ラベル名（またはプロシージャ名）
JMP     NEAR PTR    ラベル名（またはプロシージャ名）
JMP     SHORT       ラベル名（またはプロシージャ名）
```

距離のディレクティブはできるだけ付けた方が良いでしょう.

特にSHORTの場合，距離のディレクティブを付けないとNEARのコードが生成されて効率が悪くなります.

レジスタ間接ジャンプ

指定されたレジスタの値のアドレスにジャンプします.

距離はNEAR（セグメント内ジャンプ，2バイトのディスプレースメント）だけです.

実行位置は「**CS**:レジスタの値」にジャンプします.

```
JMP     BX
```

などです.

使えるレジスタは,

AX,CX,DX,BX,SP,BP,SI,DI

の8種類です.

メモリ間接ジャンプ

指定されたメモリの値のアドレスにジャンプします.

距離は次の2つです.

NEAR（セグメント内ジャンプ，2バイトのディスプレースメント）

FAR（セグメント間ジャンプ，セグメント：オフセットの4バイト

メモリの指定には，すべてのアドレッシング・モードが使用できます．

```
JMP     WORD PTR CS:[BX]
JMP     DWORD PTR ES:[BP+4]
```

などです．

必ず，

```
WORD PTR
DWORD PTR
```

のサイズ（距離）を現すディレクティブを使います．ラベル名，プロシージャ名を使わないので，こうしないと距離が分からないからです．

「レジスタ間接」と「メモリ間接」を合わせたものを，「間接ジャンプ」（インダイレクト・ジャンプ）と言っています．

レジスタ間接ではセグメント間（FAR）ジャンプが使えないこと以外はすべてのアドレッシング・モード（本書でいうメモリ/レジスタ）が使えます．

条件ジャンプ命令

8086ではZ80などと違って，「条件**CALL**」や「条件**RET**」の命令がないので，これらの条件ジャンプがプログラムを書く上で重要な要素となります．

したがって，サブルーチンなどは入口と出口が比較的固定された構造になります．

また，これらのジャンプは1バイトの範囲（+127～-128）にしかジャンプできません．

注：メモリ間接ジャンプやジャンプ・テーブルを利用したジャンプではZ80と同様またはそれ以上のテクニックが使えます．

これらの条件ジャンプは，キャリー，ゼロ，サイン，オーバーフロー，パリティのいずれか1つのフラグまたはそれらのうち複数のフラグの状態によってジャンプします．

それ以外に「**CX**レジスタが0のときジャンプする」**JCXZ**があります．

これらの条件ジャンプはサブルーチンの結果，加算，減算，比較命令などの結果でセットされたフラグによってジャンプします．

同じ条件に複数の表記法があるので注意してください．

JA
Jump if Above

JNBE
Jump if Not Below nor Equal

JAE
Jump if Above or Equal

JAC
Jump if Not Carry

JAB
Jump if Not Below

JB
Jump if Below

JC
Jump if Carry

JNAE
Jump if Not Above nor Equal

JBE
Jump if Below or Equal

JNA
Jump if Not Above

JE
Jump if Equal

JZ
Jump if Zero

JG
Jump if Greater

JNLE
Jump if Not Less nor Equal

JGE
Jump if Greater or Equal

JNL
Jump if Not Less

JL
Jump if Less

JNGE
Jump if Not Greater nor Equal

JLE
Jump if Less or Equal

JNG
Jump if Not Greater

JNE
Jump if Not Equal

JNZ
Jump if Not Zero

ニモニック	フラグ（ジャンプの条件）	意味	
JA JNBE	キャリー，ゼロが共に0	大きい	>
JAE JNC JNB	キャリー=0	大きいか等しい	>=
JB JC JNAE	キャリー=1	小さい	>
JBE JNA	キャリーまたはゼロが1	小さいか等しい	<=
JE JZ	ゼロ=1	等しい	=
JG JNLE	ゼロ=0でかつ サイン＝オーバーフロー	大きい （符号つき）	>
JGE JNL	サイン＝オーバーフロー	大きいか等しい （符号つき）	>=
JL JNGE	サイン≠オーバーフロー	小さい （符号つき）	<
JLE JNG	ゼロ=1または サイン≠オーバーフロー	小さいか等しい （符号つき）	<=
JNE JNZ	ゼロ=0	等しくない	≠
JNO	オーバーフロー=0		
JNP JPO	パリティ=0		
JNS	サイン=0		
JO	オーバーフロー=1		
JP JPE	パリティ=1		
JS	サイン=1		
JCXZ	CXレジスタ=0		

ある命令の実行結果がフラグに反映され，そのフラグの状態によりジャンプしたり，しなかったりするのがこれらの条件ジャンプ命令などですが，実際にプログラムを書いていても，すべての命令についてフラグの変化を覚えているのはなかなか困難です．

算術演算フラグは，

ゼロ・フラグ
キャリー・フラグ
補助キャリー・フラグ
オーバーフロー・フラグ
サイン・フラグ

の5つですが，補助キャリー・フラグはあまり使われることはありませんし，ジャンプでは関係ない（使わない）ので省略します．

ゼロ・フラグ
キャリー・フラグ
オーバーフロー・フラグ
サイン・フラグ

フラグに影響を与える命令を整理してみましょう．

フラグに影響を与えない命令（フラグは変化しない）

CALL	**CBW**	**CWD**	**ESC**
HLT	**IN**	条件分岐	**JCXZ**
JMP	**LAHF**	**LDS**	**LEA**
LES	**LOCK**	**LODS**	**LOOP**
MOV	**NOP**	**NOT**	**OUT**
POP	**PUSH**	**PUSHF**	**REP**
RET	**SEG**	**STOS**	**WAIT**
XCHG	**XLAT**		

すべての算術フラグに影響を与える命令

ADC	**ADD**	**CMP**	**CMPS**
NEG	**SBB**	**SCAS**	**SUB**

補助キャリー以外の算術演算フラグに影響する命令

AND	**OR**	**SAR**	**SHL**
SHR	**TEST**	**XOR**	

キャリーを除く算術演算フラグに影響を与える命令

DEC	**INC**

キャリーとオーバーフロー・フラグにのみ影響を与える命令

RCL	**RCR**	**ROL**	**ROR**

JNO
Jump if Not Overflow

JNP
Jump if No Parity

JPO
Jump if Not Parity Odd

JNS
Jump if Not Sign

JO
Jump if Overflow

JP
Jump if Parity

JPE
Jump if Parity Even

JS
Jump if Sign

JCXZ
Jump if Cx is Zero

LOOPE
LOOP if Equal

LOOPZ
LOOP if Zero

LOOPNE
LOOP if Not Equal

LOOPNZ
LOOP if Not Zero

ループ命令

ループ命令は**CX**レジスタをカウンタに使います．

まず，**CX**レジスタをデクリメントして，**CX<>0**でさらに条件によってジャンプします．また，**CX**をデクリメントした結果はフラグには影響しません．

条件ジャンプと同じ1バイト（-128～+127）の範囲へジャンプします．

これらの命令ではフラグに変化は起こりません．

ジャンプする条件

LOOP	**CX**を**-1**して，「**CX≠0**」のとき
LOOPE LOOPZ	**CX**を**-1**して，「**CX≠0**」かつ「ゼロ・フラグ=1」のとき
LOOPNE LOOPNZ	**CX**を**-1**して，「**CX≠0**」かつ「ゼロ・フラグ=0」のとき

7　プロセッサ制御命令

フラグを直接操作する命令群です．

CLC
CLear Carry

CMC
CoMplement Carry

CLD
CLear Direction flag

CLI
CLear Interrupt enable flag

STC
SeT Carry

STI
SeT Interrupt enable flag

NOP
No OPeration

CLC	キャリー・フラグをクリアする．
CMC	キャリー・フラグを反転する．「0→1」，「1→0」
CLD	ディレクション・フラグをクリアする．
CLI	インタラプト・フラグをクリアする． 外部割り込みが無効になる．
STC	キャリー・フラグをセットする．
STD	ディレクション・フラグをセットする．
STI	インタラプト・フラグをセットする． 割り込みが有効になる．
NOP	何もしない．プログラムが次に進むだけ．

8　入出力命令

入出力の単位は1または2バイトです．

acの部分は**AX**または**AL**レジスタです．

IN　　ac,DX

レジスタDXの内容で示されるI/Oポートから**ac**に読み込みます．

IN　　ac,ポート・アドレス

ポート・アドレスで示されるI/Oポートからacに読み込みます．

OUT　　DX,ac

acの内容を，**DX**レジスタの内容で示されるI/Oポートへ出力します．

OUT　　ポート・アドレス,ac

acの内容を，ポート・アドレスで示されるI/Oポートへ出力します．

9 割り込み命令

「ソフトウェア割り込み」ともいわれます．

メモリの先頭「0000:0000」から1,024（1K）バイトに記録された1つ4バイト，256個（0～255）のベクタ（アドレス）の1つにジャンプします．

このルーチンからの復帰は**IRET**命令を使います．

INT　割り込み番号

INT
INTerrupt

割り込み番号は0～255（0～FFh）のいずれかを指定します．

INTO

INTO
INTerrupt if Overflow

オーバーフロー・フラグが「1」のとき，「INT 4」を発行します．

INTO
INT 3

の2つは1バイトのコードが生成されます．他は2バイトの命令です．

INT 3は主にデバッガなどのプログラムで使われます．

IRET

IRET
RETurn from Interrupt

割り込みプログラムからの復帰は**RET**でなく必ず**IRET**を使います．

IRET命令は**CS,IP**以外にフラグ・レジスタも復元します．

10 シフト，ローテート命令

この命令は対象となるレジスタまたはメモリの内容を，「1またはCLレジスタで示されるビット数」だけ，左（ハイビットの方向）または右（ロービットの方向に）に移動します．

キャリーフラグの扱い，端のビットに入れる値によって以下の4つのグループに分けられます．

キャリーフラグを含めてリング状（ローテート）
キャリーフラグを含まずリング状（ローテート）
直線状1（論理シフト）
直線状2（算術シフト）

メモリ/レジスタの部分にはレジスタまたはメモリが指定されます．

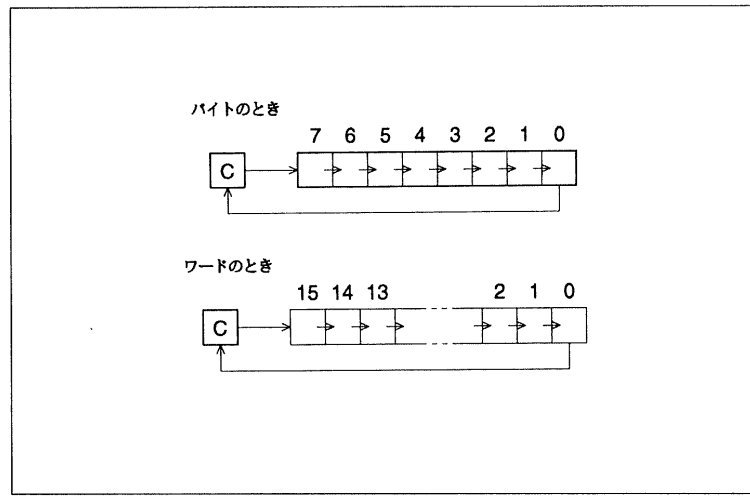
キャリーフラグを含めてリング状のパターン（ローテート）

キャリーフラグを含めてリングのような構成となります．

右に移動

RCR	メモリ/レジスタ，1
RCR	メモリ/レジスタ，**CL**

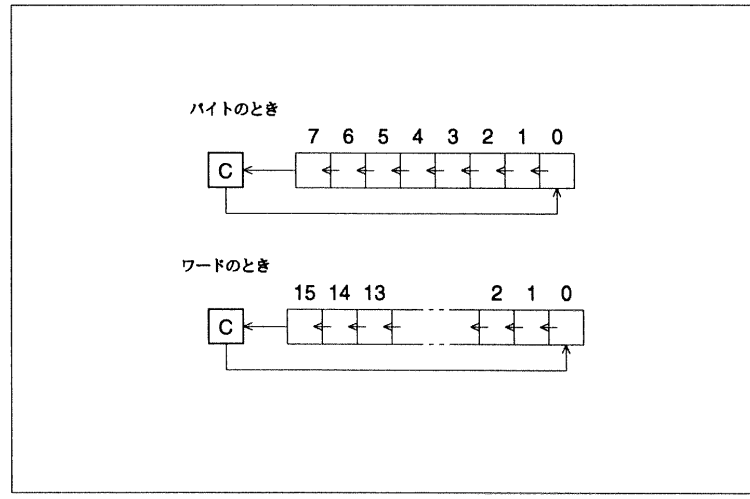
RCR
Rotate Right through Carry



左に移動

RCL	メモリ/レジスタ，**1**
RCL	メモリ/レジスタ，**CL**

RCL
Rotate Left through Carry



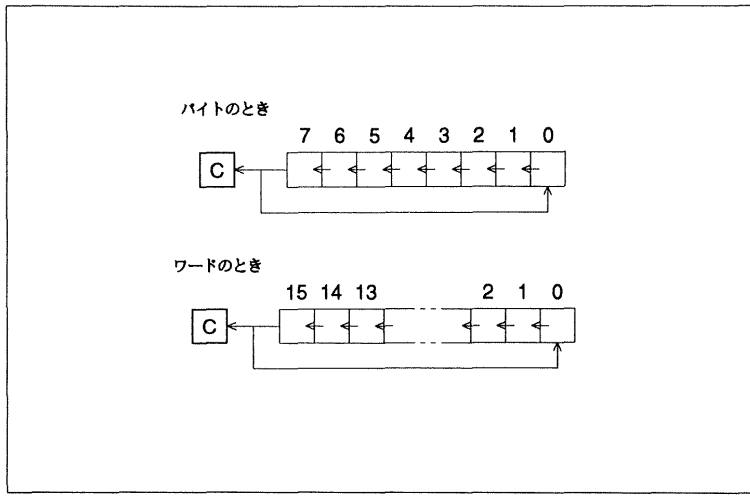
キャリーフラグを含まずリング状のパターン（ローテート）

キャリーフラグを含まずにリング状の構成となります．
キャリーフラグには最後のビットがセットされます．

左に移動

ROL	メモリー/レジスタ，**1**
ROL	メモリー/レジスタ，**CL**

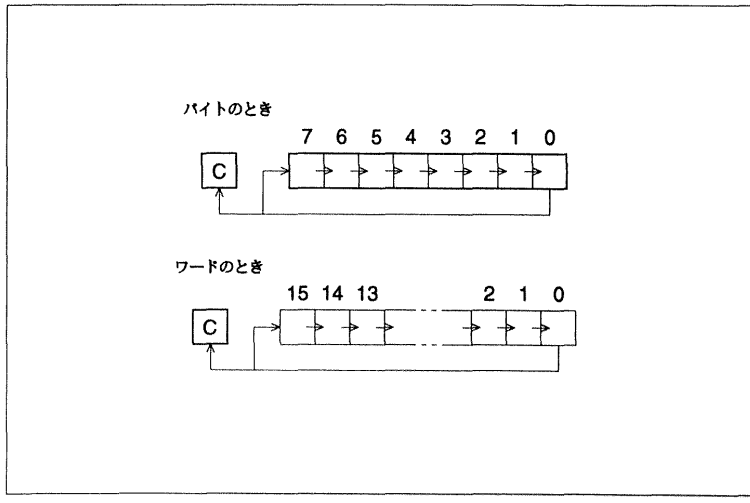
ROL
ROtate Left



右に移動

ROR	メモリー/レジスタ，**1**
ROR	メモリー/レジスタ，**CL**

ROR
ROtate Right



直線状のパターン（論理シフト）

リング状にはならないで直線状になります．
移動最初のビットには0がセットされます．

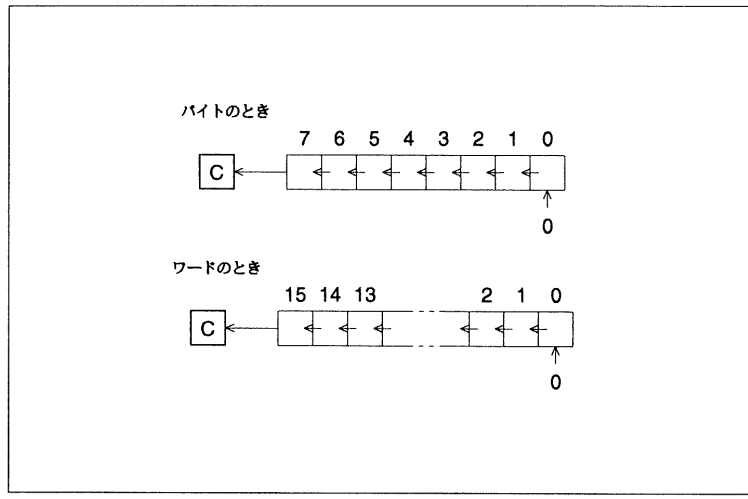
左に移動

SHL
SHift logical Left

SAL
Shift Arithmetic Left

SHL	メモリー/レジスタ，**1**
SHL	メモリー/レジスタ，**CL**

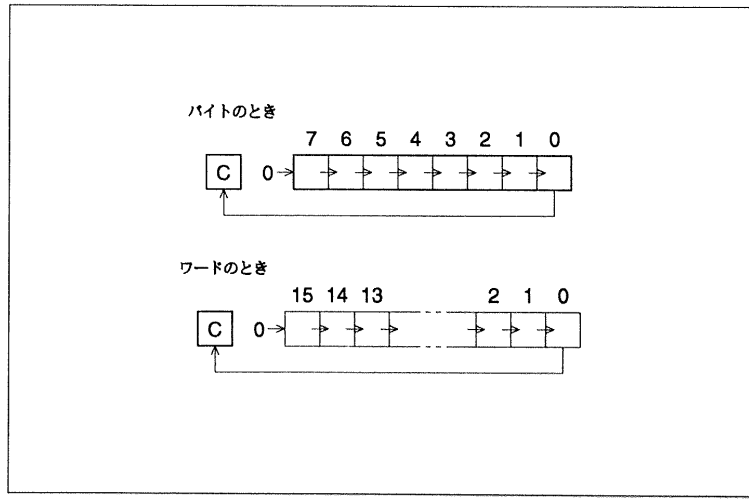
＊**SAL**　も同義



右に移動

SHR
SHift logical Right

SHR	メモリー/レジスタ，**1**
SHR	メモリー/レジスタ，**CL**



直線状のパターン（算術シフト）

リング状にはならないで直線状になります．

シフトの最初のビットは変更されません（0なら0，1なら1）．

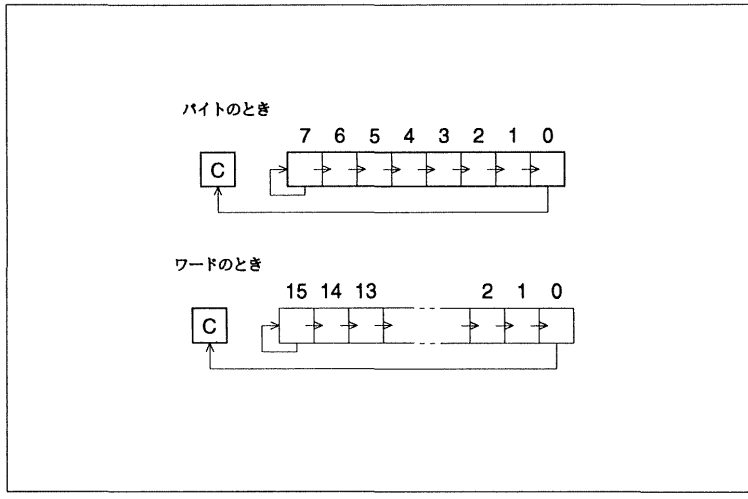
左に移動の命令はありません．

右に移動

SAR	メモリー/レジスタ，**1**
SAR	メモリー/レジスタ，**CL**

SAR
Shift Arithmetic Right

符号付きの割り算などに使えそうですね．

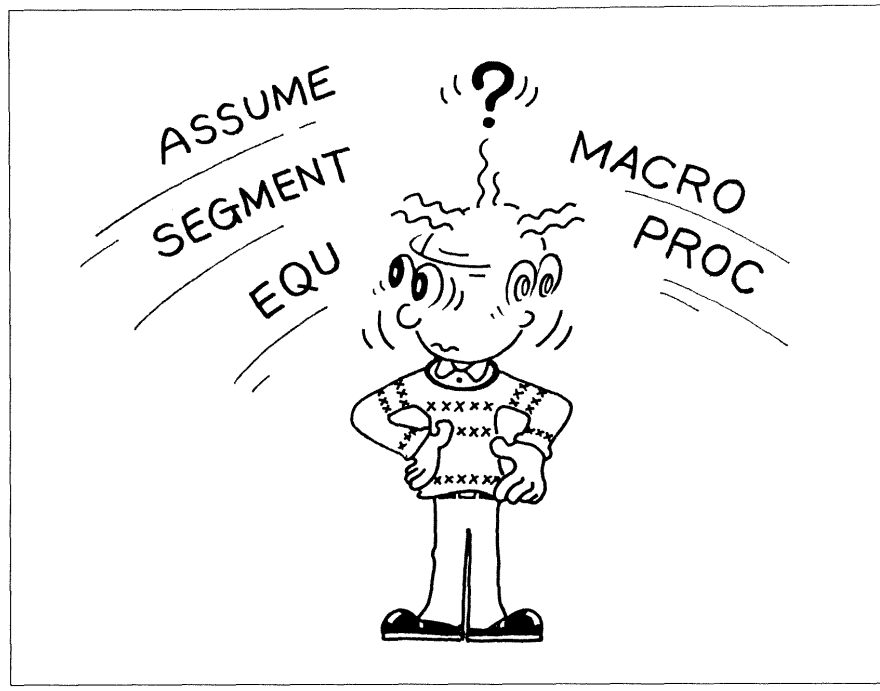


CBW CWD ESC
IN JCXZ
AHF LDS LEA
CK LODS LOOP
NOP NOT OUT
USH PUSHF REP
EG STOS WAIT

第3部　アセンブラ疑似命令を少し

第3部の説明では逆アセンブラの体験版『ZZ386ミニ』を使います．順序が前後しますが，第4部の「ZZ386の利用」を参考に『ZZ386ミニ』が動作するようにしておくとよいでしょう．

XCHG XLAT
ADD CMP CMPS
SBB SCAS SUB
OR SAR SHL
TEST XOR
INC
CR ROL ROR



ここまで来ると8086のニモニックをすべてではなくても，プログラムが書ける程度には覚えたことでしょう．しかし，残念ながらそれだけではプログラムは書けないのです．

アセンブラに指示を与えるアセンブラ命令を覚える必要があります．これも使うアセンブラ（MASM，TASM，OPTASMなど）によって若干違いますし，その種類もとても多いのです．

一般にアセンブラにCPUの種類を指示したり，リスト制御などをするものを「**アセンブラ命令**」，それ以外に**EQU**など数式記述に使ったりするものを「**ディレクティブ**」と呼ぶことが多いのですが，最近はその境界があいまいです．いっそ全部引っくるめて「**ディレクティブ**」と呼ぶことにしましょう．

使うアセンブラによっては便利なディレクティブもあるのですが，それらはゆくゆく覚えてください．ここでは最低限必要なものに付いて説明します．

1　COMファイルとEXEファイル

いきなりディレクティブの説明を始めても，なぜ必要なのかわかりません．まず全体的なイメージを理解するために機能は同じですがCOM，EXEそれぞれの実行形式用に書かれた簡単なプログラムを使って説明します．

MS-DOSにはCOM，EXEの2種類の実行可能ファイルがあります．これらは拡張子がそれぞれ「**.COM**」，「**.EXE**」であることから識別できます．大きさに注目するとCOM形式は64Kバイト以内で，EXE形式には制限がありません．

まず「**SAMPLE1.ASM**」（COM形式），「**SAMPLE2.ASM**」（EXE形式）を見て違うところを確認してください．次の3つが違います．

①EXE形式の8，9行がCOM形式にはない．

```
mov     ax,Data                 ;COMではdsのセットは不要
mov     ds,ax                   ;
```

②EXE形式ではCODE,DATA,STACKが別々に記述されている．

```
CR      equ     0dh
LF      equ     0ah

code    segment
        assume  cs:code,ds:code

        org     100h    ;ＰＳＰの次からプログラムが始まる。

start   proc    far
        mov     dx,offset Msg
        mov     ah,09h
        int     21h

        mov     ax,4c00h
        int     21h
start   endp

Msg:    db      'アセンブラ',CR,LF,'$'

code    ends
        end     start
```

```
CR      equ     0dh
LF      equ     0ah


_Prog   SEGMENT
        ASSUME  CS:_Prog, DS:_Data, SS:_STACK

start   PROC    FAR
        mov     ax,_Data                ;COMではdsのセットは不要
        mov     ds,ax
        mov     dx,offset AsmMsg
        mov     ah,09h
        int     21h

        mov     ax,4c00h
        int     21h
start   ENDP
_Prog   ENDS

_Data   SEGMENT                         ;EXEではデータセグメントが必要
AsmMsg  db      'アセンブラ',CR,LF,'$'
_Data   ENDS

_STACK  SEGMENT  STACK  'STACK'         ;EXEではスタックセグメントが必要
        dw      100h    dup(?)
_STACK  ENDS

        END     start
```

アセンブル
Assemble

リンク
Link

③COM形式の6行目がEXE形式にはない.

このソース・ファイルをアセンブル，リンクして，実行ファイルを作ってみましょう.

［例］TASMを使って作る例です.

《COM形式のとき》

```
TASM SAMPLE1
TLINK /T SAMPLE1,SAMPLE1
```

または添付ディスクの「**COM.BAT**」を使うと

```
COM SAMPLE1
```

《EXE形式のとき》

```
TASM SAMPLE2
TLINK SAMPLE2,SAMPLE2
```

または添付ディスクの**EXE.BAT**を使うと

```
EXE SAMPLE2
```

これで，それぞれ「**SAMPLE1.COM**」，「**SAMPLE2.EXE**」が作られます.

次に，「**SAMPLE1.COM**」と「**SAMPLE2.EXE**」をMS-DOSのDUMPコマンドで眺めてみましょう(「DUMP SAMPLE1.COM」で表示できます).

「**SAMPLE2.EXE**」には「4D 5A(MZ)」で始まる余分なものが付いていますね.

ヘッダ
Header

これが「EXEヘッダ」といわれるものです.

SAMPLE1.COM

```
00000000  BA 0C 01 B4 09 CD 21 B8-00 4C CD 21 83 41 83 5A   ｺ..ｴ.ｱ!ｸ.Lｱ!ｱｾ
00000010  83 93 83 75 83 89 0D 0A-24                       ﾝﾌﾗ..$
```

SAMPLE2.EXE

```
00000000  4D 5A 2D 00 02 00 01 00-20 00 21 00 FF FF 03 00   MZ-..... .!.....
00000010  00 02 29 25 00 00 00 00-1E 00 00 00 01 00 01 00   ..)%............
00000020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000200  B8 02 00 8E D8 BA 00 00-B4 09 CD 21 B8 00 4C CD   ｸ..借ｺ..ｴ.ｱ!ｸ.Lｱ
00000210  21 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   !...............
00000220  83 41 83 5A 83 93 83 75-83 89 0D 0A 24            ｱｾﾝﾌﾗ..$
```

このようにCOMとEXEではファイルの形式が違います．またプログラムがメモリに読み込まれる様子が少し違います．

ファイル中の実際にメモリに読み込まれる（ロードされる）部分を「**ロード・モジュール**」といいます．

ロード・モジュール
Load Module

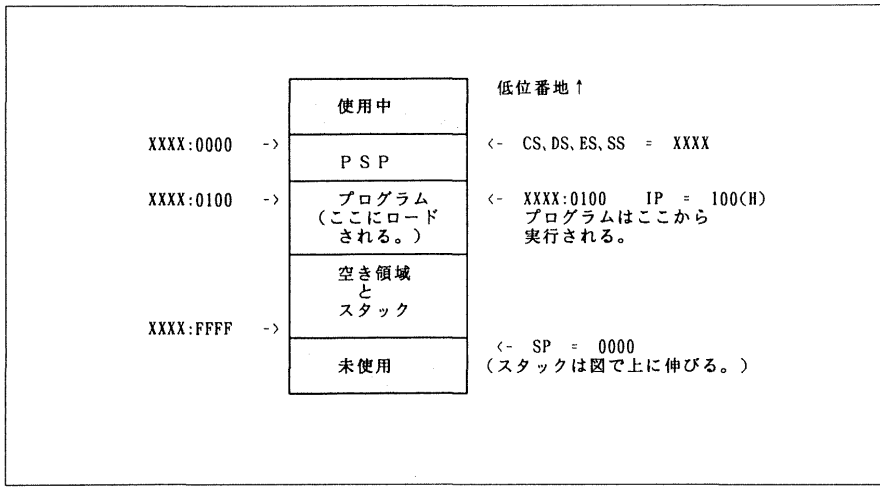
COMファイルはロード・モジュールだけで構成されています．

EXEファイルは「EXEヘッダ＋ロード・モジュール」という構成です．ロード・モジュールはメモリに読み込まれた後，EXEヘッダの情報にもとづき一部（セグメント値の部分）が変更されます．一部のレジスタもヘッダ情報によってセットされます．

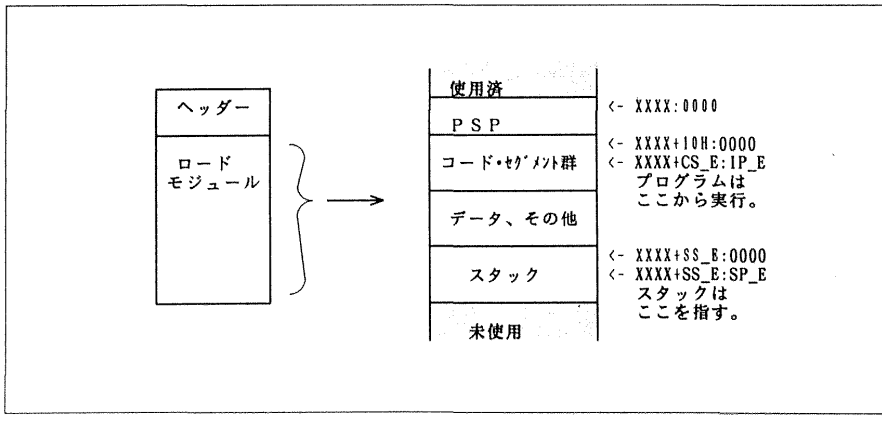
COM形式の実行ファイルとEXE形式の実行ファイルがどのようにパソコンのメモリに読み込まれるのかを少し考えてみましょう．

簡単には，「COMは小さい」，「EXEは大きい」といえるかも知れません．

COMのロードイメージ



EXEのロードイメージ



COM形式ではファイルがまるごとメモリにコピーされます．EXE形式ではロード・モジュールだけがメモリに読み込まれますが，COMファイルはロード・モジュールだけで構成されているといえます．

リロケーション
Relocation

EXEファイルのロード・モジュール以外の部分，「EXEヘッダ」という部分にはリロケーション・データ，レジスタの初期値などが記録されています．

リロケーション・データは，

```
jmp     セグメント：オフセット
mov     ax, SEGMENT SEG DATA
```

などの「＿＿の部分」，セグメント値を扱っているところを，プログラムが読み込まれた状態で正しい値に調整するための情報です．

COMとEXEの違いを整理してみましょう．

	COM形式	EXE形式
最大サイズ	65,536バイトからPSP分256バイトとスタック分2バイトを引いた量	制限なし
エントリ・ポイント	PSP:100H	ENDステートメントによって決まる．
エントリでの**CS**	PSP	プログラムのエントリ・ポイントを含むセグメント
エントリでの**IP**	0100H	セグメント中のエントリ・ポイントのオフセット
エントリでの**DS**	PSP	PSP
エントリでの**ES**	PSP	PSP
エントリでの**SS**	PSP	**STACK**属性をもつセグメント
エントリでの**SP**	FFFEHと使用可能メモリの最高位アドレスのどちらかで，値の低いほう	**STACK**属性をもつセグメントの最後
エントリでのスタック	0ワード	ソースに依存
スタックのサイズ	65,536バイトからPSP分256バイトとコードのサイズを引いた量	**STACK**属性をもつセグメントで決まる
サブルーチン・コール	NEAR	NEARまたはFAR
ファイルのサイズ	プログラムのサイズ	プログラムのサイズにヘッダ（最低512バイト）を足したもの

ＰＳＰ

COMであれ，EXEであれ，プログラムがメモリに読み込まれ実行されようとするときにはPSPがそのプログラムの先頭に付加されます．PSPはもとの（実行しようとする）プログラムには含まれていません．

PSPはDOSがプログラムにパラメータなどの必要な情報を渡すのに使われます．このことはDOS上で実行するプログラムの決まりですから，不便であっても仕方ありません．

PSPの内容

アドレス	内　容
00h～01h	INT20hのアドレス
02h～03h	メモリの最上位
04h～09h	MS-DOS予約
0Ah～0Dh	終了アドレス
0Eh～11h	Ctrl-C出口アドレス
12h～15h	致命的エラー・ハンドラのアドレス
16h～2Bh	MS-DOS予約
2Ch～2Dh	環境ポインタ
2Eh～4Fh	MS-DOS予約
50h～51h	DOSコール
52h～5Bh	MS-DOS予約
5Ch～6Bh	第１FCB（非オープン）
6Ch～7Fh	第２FCB（非オープン）
80h	パラメータの長さ（バイト数）
81h～FFh	コマンド・パラメータ

疑似命令「SEGMENT」

ディレクティブの最初の難関はセグメントに関する記述です．

SEGMENT疑似命令について最初におちいりやすい誤解は，**SEGMENT**疑似命令で定義する「セグメント」とCPUの各セグメント・レジスタが指すメモリ空間の「セグメント」を混同して，どうしてもメモリ空間の「セグメント」の認識に引かれやすいのです．第1にリンクをイメージしてください．

ソース・プログラムに記述するセグメントはリンクの最小単位と考えてください．

Z80の組み込みプログラムなどを書いたことのある人なら容易に想像がつくと思いますが，「コードと初期化ずみデータはROMに，作業用データはRAMに置いて」などとと考えますよね，グループ化と配置を考慮することが必ず必要でした．

8086でも組み込みプログラムを書くときは順序，配置（アドレス）を当然考える必要があります．

しかし，MS-DOSで動くプログラムを書くのにそんなことを考える必要があるのでしょうか．

ここで思い出してください．8086は各セグメントレジスタが指す4つのそれぞれ64Kバイトの空間を同時にもつことができます．またその4つの空間を複数もつこともできます．

では，「**CS=DS=ES=SS**」となり，1つの64Kバイトの空間しかもたないCOMモデルのプログラムではどうでしょうか．

そうです．そんな面倒なことは何も考える必要はありません．

もう1度，「**SAMPLE1.ASM**」を見てください．

「**code segment**」と「**code ends**」の間にすべてを詰め込んでしまえばすむのです．

「当面はCOMモデル以外のプログラムは書かないし，複数のソース・ファイルからなる大きなプログラムも書かない」という人は次の説明にジャンプしてください．

```
code    segment                 ←ここから始まる．
        org     100h            ;ＰＳＰの次からプログラムが始まる．
start   proc    far
        mov     dx,offset Msg
        mov     ah,09h
        int     21h

        mov     ax,4c00h
        int     21h
start   endp

Msg:    db      'アセンブラ',0dh,0ah,'$'
code    ends            ←ここで終わる．
```

もう1度「セグメントはリンクするときの単位」ということを確認してください．

セグメントには「セグメント名」と「クラス名」を付けることができます．

クラス
Class

セグメントはそのセグメント名とクラス名で管理されます．

セグメント名が評価され，次にクラス名が評価されます．

特に理由がない限り，セグメント名，クラス名をともに使うほうがよいと思います．

リンカは同じ名前のセグメントを同じ名前のセグメントの後ろに結合します．

次に同じクラス名のセグメントが結合されます．

最後に，それぞれのクラスをリンカに読み込まれた順番に結合します．

したがって，リンク時のファイル指定の順序が違うと変な配置になったりもします．

そのため，C言語のスタートアップ・ルーチン（最初にリンクされるルーチン）では空のセグメントを定義し，いつも同じ順序になるようになっています．

これほど沢山のクラスは必要ないと思いますが，いつも先頭のファイルでこのように配置を決めるのは良い習慣だと思います．

Cスタートアップ・ルーチンのマップ・ファイルの例

```
;Segment and Group declarations

  _TEXT                   SEGMENT BYTE PUBLIC 'CODE'
                          ENDS
  _FARDATA                SEGMENT PARA PUBLIC 'FAR_DATA'
                          ENDS
  _FARBSS                 SEGMENT PARA PUBLIC 'FAR_BSS'
                          ENDS
  IFNDEF __TINY__
  _OVERLAY_               SEGMENT PARA PUBLIC 'OVRINFO'
                          ENDS
  _1STUB_                 SEGMENT PARA PUBLIC 'STUBSEG'
                          ENDS
  ENDIF
  _DATA                   SEGMENT PARA PUBLIC 'DATA'
                          ENDS
  _INIT_          SEGMENT WORD PUBLIC 'INITDATA'
  InitStart               label byte
                          ENDS
  _INITEND_               SEGMENT BYTE PUBLIC 'INITDATA'
  InitEnd         label byte
                          ENDS
  _EXIT_                  SEGMENT WORD PUBLIC 'EXITDATA'
  ExitStart               label byte
                          ENDS
  _EXITEND_               SEGMENT BYTE PUBLIC 'EXITDATA'
  ExitEnd                 label byte
                          ENDS
  _CVTSEG                 SEGMENT WORD PUBLIC 'DATA'
                          ENDS
  _SCNSEG                 SEGMENT WORD PUBLIC 'DATA'
                          ENDS
  IFNDEF __HUGE__
    _BSS                  SEGMENT WORD PUBLIC 'BSS'
                          ENDS
    _BSSEND     SEGMENT BYTE PUBLIC 'BSSEND'
                          ENDS
  ENDIF
  IFNDEF __TINY__
   _STACK                 SEGMENT STACK 'STACK'
                  ENDS
  ENDIF
```

sample2（EXE）のマップ・ファイル

```
 Start  Stop   Length Name           Class

 00000H 00012H 00013H _CODE
 00020H 0002CH 0000DH _DATA
 00030H 0012FH 00100H _STACK

Program entry point at 0000:0000
Warning: No stack
```

SEGMENT疑似命令の使い方

セグメント名 **SEGMENT [align] [combine] [use] ['class']**

[ ]に囲まれているパラメータは省略可能.

セグメント名は必ず記述します.

“**use**”は80386のコードを出力するときに使用するパラメータで一般にはあまり使いません. デフォルトが16ビットか32ビットを指定します.
USE16, **USE32**のいずれかです.

それ以外のパラメータは必要なので覚えてください.

align…アラインの指定

セグメントの開始アドレスの調整の指定です. 調整の方法は, 次のいずれかです.

BYTE	調整しない.
WORD	ワード境界（偶数番地）から始まる.
DWORD	ダブル・ワード境界（4の倍数番地）から始まる.
PARA	パラグラフ境界（16の倍数番地）から始まる.
PAGE	ページ境界（256の倍数番地）から始まる.

何も指定しないと**PARA**になります.

このパラメータは, セグメントが結合されるとき, 指定された境界から始まるように調整されます（スキ間が空く）.
“**BYTE**”を指定すると調整されずに, 密着して結合されます.

このパラメータの必要性はいろいろと考えられますが, MS-DOSのプログラムを考えたとき問題になるのは,

①奇数アドレスから始まるワード（2バイト）にアクセスすると時間が余分にかかる.
386以上（特に486）のCPUでキャッシュを使ったときなど, どうなるのか私は良く知りません.
②データセグメント・レジスタを切り替えてアクセスするデータ, スタックなどはセグメント境界「PARA」に置いたほうが良い.

などが考えられます. このような理由でいちばん安全な「**PARA**」がデフォルトになっています.

combine…セグメントの結合指定

セグメントの結合の指定をします．

PUBLIC	同じ名前のセグメントがあれば結合する．
MEMORY	**PUBLIC**と同じ
PRIVATE	どのセグメントとも結合されない（独立したセグメントになる）． 何も指定しない場合はこの型になる．
STACK	リンカはこのセグメントの先頭を**SS**に，このセグメントのサイズを**SP**にセットする． 実際はEXEヘッダに記録される．したがってCOMモデルでは使えない．
AT　パラグラフ	セグメントの絶対アドレスをパラグラフで指定する． コード，データは生成されない．
COMMON	同じ名前のセグメントをすべて同じアドレスから配置する． 1番大きなセグメントのサイズがそのサイズになる．
VIRTUAL	TASM専用．ほとんど使いません．

通常使われるのは，**PUBLIC**，**PRIVATE**，**STACK**です．

絶対アドレスを指定してコード，データを出力する命令がないので，ROM化などをするときは「リンケージ・エディタ」というソフトが使われます．

class…クラス名の指定

クラス名は'または"で囲んで記述します．

同じクラス名をもつセグメントは連続して配置されます．

セグメントの配置はリンカにマップ・ファイルを出力させて確認することができます．Tlinkでは「**/m**」または「**/s**」のオプションを指定します．

マップ・ファイルの例

アドレス		大きさ	セグメント名	クラス名
↓	↓	↓	↓	↓
Start	Stop	Length	Name	Class
00000H	0129FH	012A0H	_TEXT	CODE
012A0H	012A0H	00000H	_FARDATA	FAR_DATA
012A0H	012A0H	00000H	_FARBSS	FAR_BSS
012A0H	012A0H	00000H	_OVERLAY_	OVRINFO
012A0H	012A0H	00000H	_1STUB_	STUBSEG
012A0H	01591H	002F2H	_DATA	DATA
01592H	01592H	00000H	_CVTSEG	DATA
01592H	01592H	00000H	_SCNSEG	DATA
01592H	01592H	00000H	_CONST	CONST
01592H	01597H	00006H	_INIT_	INITDATA
01598H	01598H	00000H	_INITEND_	INITDATA
01598H	01598H	00000H	_EXIT_	EXITDATA
01598H	01598H	00000H	_EXITEND_	EXITDATA
01598H	015D9H	00042H	_BSS	BSS
015DAH	015DAH	00000H	_BSSEND	BSSEND
015E0H	0165FH	00080H	_STACK	STACK

結論

COMモデルでソース・プログラム（「**.asm**」ファイル）が1本であれば何も考えることはありません．おまじないのように次のスタイルでソース・ファイルを書けばばよいのです．

```
CODE    SEGMENT
        org   100h
先頭のラベル名（またはプロシージャ）

    この間にプログラムを書く　・・・・

CODE    ENDS
        END  先頭のラベル名（またはプロシージャ）
```

2 複数のソース・ファイルから構成されるCOMファイル

では，簡単なソース・ファイル2本から構成されるプログラムの例を説明しましょう．

ここでは共通となる定義部分はファイル「**SAMPLE3.INC**」にまとめられ，各ソース・ファイルからインクルード（読み込み）されています．共通の定義などを1箇所で処理，修正などが出来るのでトラブルが避けられます．

共通の定義ファイルは3つの部分からなりますが，最初がセグメントの定義です．2番目がグループの定義です．最後が定数の定義ですが，これはここでは関係ないので説明は省略します．

まずセグメントの定義部分，ここではセグメント名，アライメント，結合規則，クラス名が定義されています．**CODE**以外の「**_CODE**」クラスのセグメント名，**DATA**以外の「**_DATA**」クラスのセグメント名は定義されていないので，セグメント名が**CODE**，**DATA**という順序でリンクされます．

次にグループの定義部分，1つのグループに指定されたセグメントは同じセグメント値と見なされます．

グループを指定しておくと，

```
mov     dx, offset CGroup:Msg      <-sample31.asm
mov     dx, offset CGroup:Msg2     <-sample32.asm
```

などのようにグループ名を使ってアクセスできるようになります．

当然ですが，グループに指定されたセグメントは合計で64Kバイトを越えることはできません．

「**SAMPLE31.ASM**」，「**SAMPLE32.ASM**」が各ソース・ファイルです．

マップ・ファイルで確認するため，シンボルは必要以上に**public**宣言されています．「**SAMPLE31.MAP**」を見てリンクのイメージを確認してください．

sample3.inc

```
;-------------------------------------------------------------------
;       インクルードファイル
;       ソースファイル          sample31.asm    asmple32.asm
;       実行ファイル            sample31.com
;       実行ファイル作成バッチファイル          s3.bat
;-------------------------------------------------------------------

;-----------------------------------------------------
;       セグメント名の定義
;       これによりリンクの順序が決まる
;-----------------------------------------------------
_CODE   SEGMENT BYTE    PUBLIC  'CODE'
_CODE   ENDS
_DATA   SEGMENT BYTE    PUBLIC  'DATA'
_DATA   ENDS

;-----------------------------------------------------
;       同じセグメントレジスタ値でアクセス出来るように
;       グループ化する
;-----------------------------------------------------
CGroup  GROUP   _CODE, _DATA


;-----------------------------------------------------
;       キャリッジリターンの定義
;-----------------------------------------------------
CR      equ     0dh
LF      equ     0ah
```

sample31.asm

```
        include         sample3.inc

extrn   start2  :       near

        ASSUME  cs:CGroup, ds: CGroup

_CODE   SEGMENT
        org     100h    ;PSPの次からプログラムが始まる。

public  start
start   PROC    NEAR
        mov     dx,offset CGroup:Msg
        mov     ah,09h
        int     21h
        jmp     start2
start   ENDP
_CODE   ENDS

_DATA   SEGMENT BYTE
public  Msg
Msg:    db      'アセンブラ１',CR,LF,'$'
_DATA   ENDS

        end     start
```

sample32.asm

```
        include         sample3.inc

public  start2

        ASSUME  cs:CGroup, ds: CGroup

_CODE   SEGMENT

public  start2
start2  PROC    NEAR
        mov     dx, Offset      CGroup:Msg2
        mov     ah,09h
        int     21h

        mov     ax,4c00h
        int     21h
start2  ENDP

_CODE   ENDS

_DATA   SEGMENT
public  Msg2
Msg2:   db      'アセンブラ２',CR,LF,'$'
_DATA   ENDS

        END
```

sample31.map

```
 Start  Stop   Length Name               Class

 00000H 00115H 00116H _CODE             CODE
 00116H 00133H 0001EH _DATA             DATA


Detailed map of segments

 0000:0000 010A C=CODE   S=_CODE         G=CGROUP  M=SAMPLE31.ASM ACBP=28
 0000:010A 000C C=CODE   S=_CODE         G=CGROUP  M=SAMPLE32.ASM ACBP=28
 0000:0116 000F C=DATA   S=_DATA         G=CGROUP  M=SAMPLE31.ASM ACBP=28
 0000:0125 000F C=DATA   S=_DATA         G=CGROUP  M=SAMPLE32.ASM ACBP=28

  Address         Publics by Name

 0000:0116 idle  MSG
 0000:0125 idle  MSG2
 0000:0100 idle  START
 0000:010A       START2

  Address         Publics by Value

 0000:0100 idle  START
 0000:010A       START2
 0000:0116 idle  MSG
 0000:0125 idle  MSG2
```

この2つのソース・ファイルから実行ファイルを作るためのバッチ・ファイル「**S3.BAT**」を用意しています．

S3.BAT

tasm /l sample31	/l はリスト・ファイルを出力するオプション
tasm /l sample32	
tlink /s /t sample31 sample32	/sは詳細なマップ・ファイルを出力するオプション /tはCOMファイルを出力するオプション

```
A>s3
```

と入力すると，実行ファイル「**SAMPLE31.COM**」が作成されます．

ソース・ファイルの一部を次のように変更して生成されたのが「**SAMPLE3B.COM**」です.

```
mov   dx, offset Msg          <-sample31.asm
mov   dx, offset Msg2         <-sample32.asm
```

```
A>sample31
```

とタイプして「**SAMPLE31.COM**」を実行すると,

```
アセンブラ1
アセンブラ2
```

が表示されます.

```
A>sample3b
```

とタイプしてSAMPLE3B.COMを実行すると，ゴミ（意味のない文字）が数行表示されます.

「**SAMPLE3B.COM**」では文字列が正しくセットされていないのが想像されますが，なぜこうなったのか，アセンブラによって出力されたリスト・ファイルを比べて調べてみましょう.

```
正常：44    0100 BA 0000r    mov   dx,offset CGroup:Msg
異常：45    0100 BA 0000r    mov   dx,offset Msg
```

これからでは違いは分かりませんね.

では『ZZ386ミニ』を使って実行ファイル確認してみましょう．それぞれ,

```
A>zz386 s3.zpj          ←正：sample31.comの逆アセンブル
A>zz386 s3b.zpj         ←誤：sample3b.comの逆アセンブル
```

いずれも「ZZ386ミニ」が起動したら，[M], [3]とキーを押してください.
実行ファイルが逆アセンブルされ，画面に表示されます.
終了するときは[f・1], [Q], [Y]とキーを押してください.

図zz386-1，図zz386-2がそれぞれ**SAMPLE31.COM**，**SAMPLE3B.COM**の逆アセンブル結果です．図zz386-3，図zz386-4 がそれぞれのプロジェクト・ファイルの内容です.

それぞれの逆アセンブル結果を比べると，出力する文字列のアドレスを**DX**レジスタにセットしている部分が違いますね．**SAMPLE3B.COM**では正しくセットされていません.

```
誤：C1000:0100    PRG_START:    MOV   DX,00006H    ;BA 0006
誤：C1000:010A    CI_1010A:     MOV   DX,00015H    ;BA 0015
```

このように，アセンブルおよびリンクが正常に（エラーなどが検出されずに）終わるがプログラムは正常に動作しない，ということがアセンブラでプログラムを書くときはよく起こります.

この場合はデータのセット部分のミスなので，プログラムの暴走（コンピュータをリセットしないといけない状態）にまでは至りませんでしたが，コードに関することであればまず暴走していたでしょう．最悪の場合はディスクの破壊まで起こしかねません．

もっとも，私の経験から言って，COMモデルやスモール・モデルのプログラムの開発では，そこまで至ることはまずありません．でも，念のためにも開発中はバックアップを頻繁に取ることをおすすめします．

このようなトラブルを未然に防ぐためにいつも同じスタイルでプログラムを書くことがよいでしょう．セグメントに関するディレクティブ（疑似命令）について整理してみましょう．

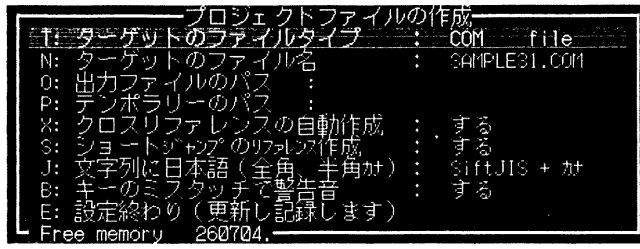
---

①リンクする順序にしたがいセグメント名を定義する．
②セグメントをグループ化する．このときメモリ・モデルを考慮します．
　COMモデルならすべてが同じセグメントなので同一グループとします．
　スモール・モデルならコード，データ，スタックのグループとします．
　ラージ・モデルなら複数のコード，データと1つのスタックとします．
③データなどへのアクセスではグループ名を基準に（タグとして）アクセスします．

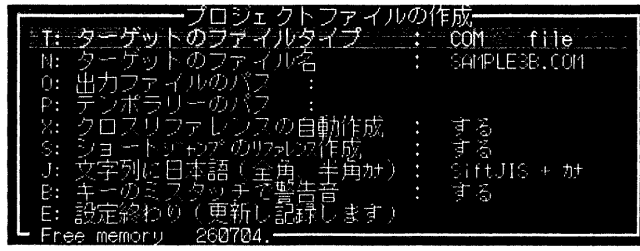
図zz386-1

図zz386-2

図zz386-3



図zz386-4



3 複数のソース・ファイルから構成されるEXEファイル

EXEファイルの最大の特徴は64Kバイトというファイル・サイズの制限がないことです.

そのため大きなプログラムや大きなデータを扱うことができます.

SAMPLE3と同じ機能でEXEモデルに書き直したのが「**SAMPLE41.ASM**」,「**SAMPLE42.ASM**」,「**SAMPLE4.INC**」です.

「**S4.BAT**」で実行ファイル「**SAMPLE41.EXE**」が作られます.

まず違っているのは，「**SAMPLE4.INC**」の**GROUP**の記述がなくなっていますね．

コードとデータを同じセグメントにするとEXEファイルにする意味がないからです．

その代わり，スタック・セグメントの記述が追加されています．EXEモデルではコード，データ，スタックは別々のセグメント（空間）を使うからです．

SAMPLE31の先頭でDSにデータ・セグメント（**_DATA**）の値をセットしていますね．

```
mov    ax, seg _DATA
mov    ds, ax
```

EXEファイルでは「**CS:IP**」と「**SS:SP**」はEXEヘッダに記録されていて，プログラムが実行されるときには適切な値にセットされています．

一方，**DS**と**ES**は**PSP**を指していて，DSはデータ・セグメント（**_DATA**）を指してはいません．そのためこのような処理が必要になります．

また文字列のアドレスをセットしている部分では「**_DATA**」を基準としています．

```
mov    dx, offset _DATA:Msg          ← sample41.asm
mov    dx, offset _DATA:Msg2         ← sample42.asm
```

マップ・ファイルで，その結果を確認してください．

『ZZ386』で確認するために「**S4.ZPJ**」を用意しています．

```
;-------------------------------------------------------------
;       インクルードファイル
;       ソースファイル          sample41.asm    asmple42.asm
;       実行ファイル            sample41.exe
;       実行ファイル作成バッチファイル          s4.bat
;-------------------------------------------------------------

;-------------------------------------------------------
;       セグメント名の定義
;       これによりリンクの順序が決まる
;-------------------------------------------------------
_CODE   SEGMENT BYTE    PUBLIC  'CODE'
_CODE   ENDS
_DATA   SEGMENT BYTE    PUBLIC  'DATA'
_DATA   ENDS
_STACK  SEGMENT STACK 'STACK'
_STACK  ENDS

;-------------------------------------------------------
;       キャリッジリターンの定義
;-------------------------------------------------------
CR      equ     0dh
LF      equ     0ah
```

```
        include         sample4.inc

extrn   start2  :       near

        ASSUME  cs:_CODE, ds: _DATA

_CODE   SEGMENT BYTE    PUBLIC

public  start
start   PROC    NEAR
        mov     ax, seg _DATA
        mov     ds, ax
        mov     dx,offset _DATA:Msg
        mov     ah,09h
        int     21h
        jmp     near ptr start2
start   ENDP
_CODE   ENDS

_DATA   SEGMENT BYTE
public  Msg
Msg     db      'アセンブラ１',CR,LF,'$'
_DATA   ENDS

_STACK  SEGMENT STACK 'STACK'
        DW      100h    DUP(?)
_STACK  ENDS

        END     start
```

```
        include         sample4.inc

public  start2

        ASSUME  cs:_CODE, ds: _DATA

_CODE   SEGMENT BYTE    PUBLIC

public  start2
start2  PROC    NEAR
        mov     dx, offset _DATA:Msg2
        mov     ah,09h
        int     21h

        mov     ax,4c00h
        int     21h
start2  ENDP

_CODE   ENDS

_DATA   SEGMENT BYTE    PUBLIC
public  Msg2
Msg2    db      'アセンブラ２',CR,LF,'$'
_DATA   ENDS

        END
```

```
 Start  Stop   Length Name                   Class

 00000H 0001AH 0001BH _CODE                 CODE
 0001BH 00038H 0001EH _DATA                 DATA
 00040H 0023FH 00200H _STACK                STACK


Detailed map of segments

 0000:0000 000F C=CODE   S=_CODE            G=(none)  M=SAMPLE41.ASM ACBP=28
 0000:000F 000C C=CODE   S=_CODE            G=(none)  M=SAMPLE42.ASM ACBP=28
 0001:000B 000F C=DATA   S=_DATA            G=(none)  M=SAMPLE41.ASM ACBP=28
 0001:001A 000F C=DATA   S=_DATA            G=(none)  M=SAMPLE42.ASM ACBP=28
 0004:0000 0200 C=STACK  S=_STACK           G=(none)  M=SAMPLE41.ASM ACBP=74
 0004:0200 0000 C=STACK  S=_STACK           G=(none)  M=SAMPLE42.ASM ACBP=74

  Address         Publics by Name

 0001:000B idle  MSG
 0001:001A idle  MSG2
 0000:0000 idle  START
 0000:000F       START2

  Address         Publics by Value

 0000:0000 idle  START
 0000:000F       START2
 0001:000B idle  MSG
 0001:001A idle  MSG2

Program entry point at 0000:0000
```

A>ZZ386 S4.ZPJ

で確認してみてください.

『ZZ386』が起動したら, M 3 とキーを押してください.
「**SAMPLE41.EXE**」が逆アセンブル表示されます.
コード, データのセグメントが別々になっているのが確認できます.

```
ZZ386 ミニ版 for 8086. Ver 1.0 (c)Micro-org 1992 [S4.ZPJ]          -8086-
        START_SEG Segment Byte Public
        Assume CS:START_SEG , DS:START_SEG , SS:STACK_SEG
        SAMPLE41  proc  Far
        プログラムの実行開始点
C1010:0000PRG_START:MOV    AX,Seg Seg_3                 ;B8 1011
C1010:0003          MOV    DS,AX                        ;8E D8
        Assume DS:Seg_3
C1010:0005          MOV    DX,Offset ES_1011B           ;BA 000B
C1010:0008          MOV    AH,009H                      ;B4 09
        文字列の出力 AH=09 文字列(DS:DX)-->
C1010:000A          INT    021H                         ;CD 21
C1010:000C          JMP    Near Ptr CI_1010F            ;E9 0000
C1010:000FCI_1010F: MOV    DX,Offset ES_1012A           ;BA 001A
C1010:0012          MOV    AH,009H                      ;B4 09
        文字列の出力 AH=09 文字列(DS:DX)-->
C1010:0014          INT    021H                         ;CD 21
C1010:0016          MOV    AX,04C00H                    ;B8 4C00
        子プロセスの終了 AH=4C AL=Ret Code.
C1010:0019          INT    021H                         ;CD 21
        SAMPLE41  endp
        START_SEG Ends
        Seg_3 Segment Byte Public
        Assume CS:Seg_3 , DS:Seg_3 , SS:STACK_SEG
 End   Option Coment Label  Jump        Search segmnt X-fer  Inasm  Make
```

逆アセンブル＆再現の困難①

画面に表示するときはスタック・セグメントは表示されません．プログラム中にあるときは表示されますが，普通，スタックはEXEヘッダにその位置と大きさが記録れているだけでロード・モジュールには含まれていません．

逆アセンブル結果をファイルに出力したときはスタック・セグメントは出力されます．

SAMPLE41.EXEを逆アセンブルし，それをもとに実行ファイルを作ってみましょう．

ソース形式のファイルを出力し，それをアセンブルして**S4X.EXE**を作ってみます．

操作は次のとおりです．

M 3	自動解析をする．
U	逆アセンブル出力．
O A	オプションをソース形式にする

＊ポップアップ・ウィンドウが「ソース形式」になっているのを確認

ESC	逆アセンブル出力．（コマンドレベルを１つ戻る）
F	ファイルに出力
A	すべてを出力する．

その後，ファイル名を確認するのでRETURNキーを2回押します．

「**S4X.ASM**」と「**S4X.EQU**」が出力されます．「**S4X.EQU**」にはこの場合なにも書かれていませんが，とりあえず消さないでください．

「**S4XX.BAT**」を使ってアセンブル，リンクをして「**S4X.EXE**」を作ってみましょう．

「**S4XX.BAT**」の内容は

```
tasm /l s4x
tlink /s s4x
```

逆アセンブルからできたファイル「S4X.EXE」と元の「**SAMPLE41.EXE**」を較べてみましょう．

```
fc s4x.exe sample4 1.exe
```

と入力してください．fcはMS-DOSのファイル比較コマンドです．

次のように表示されます．

```
ファイルの比較中です S4X.EXE と SAMPLE41.EXE
0000020C: EB E9
0000020D: 01 00
0000020E: 90 00
```

なぜ同じものができないのでしょうか．

リスト・ファイル「**S4X.LST**」を見てみましょう．

20C: 20D: 20E:の部分は次の行に対応します．

```
21 000C EB 01 90 JMP    NEARPTR CI_1010F   ;E9 00 00
```

よく見ると，「EB 01 90」となっています．これは次の命令のコードです．

```
JMP SHORT CI_1010F
NOP
```

アセンブラが勝手に命令を変えているのですね．ただし，アセンブラによってはこのような変換をしないものもあります．

元のソース・ファイルは複数（この場合2本）から構成されています．

一方，逆アセンブルで得られたソース・ファイルは1本です．このことが色々な違い引き起こすので留意していてください．

*

このことは，386以上のコードでネイティブ（386本来の）モードとリアル（8086互換）モードが混在した実行ファイルを逆アセンブルするときにも問題になります．

私の知る限りでは，1つのソース・ファイル中で，リアル・モードのコードととネイティブ・モードのコードが混在できるアセンブラはないと思っています．

ベースが16ビットで一部32ビットの命令を使うとか，その逆はできますが，ベース自体を途中で変えることはできません．

開発するときは当然リアル・モードの部分とネイティブの部分は別のソース・ファイルという構成になっているでしょう．逆アセンブルにおいてもそのような工夫が必要になります．

4 プログラムの単位＝プロシージャ

「プロシージャ」はプログラムの最小単位と考えてください．サブルーチンと考えても差し支えありません．

```
name  PROC [ language modifier ] [ language ] [ distance ][ USE item ]
      [ argument [, argument ] ] [ RETURNS argument [, argument ] ]

この間にプログラム（プロシージャの中身）を書きます．．．．

name  ENDP
```

大文字の部分は予約語です．[ ]に囲まれた部分は省略できますが，それ以外は省略できません．**PROC**の行で始まり，**ENDP**の行で終わります．その間にプログラムを記述します．

language modifier：　省略可

このパラメータはプロシージャの入口と出口で，MS-Windows用のスタック・セットアップのコードを自動生成するためなどのものなので，ここでは触れません．

language:　省略可

NOLANGUAGE，**C**，**PASCAL**，**BASIC**，**FORTRAN**，**PROLOG**のいずれかが使えます．

他言語から呼び出されるプロシージャ（サブルーチン）などを作るとき，プロシージャの入口と出口でスタック・セットアップのコードを自動生成するためのパラメータです．通常のアセンブラ・プログラミングでは使いません．

name:　省略不可

プロシージャの名前です．他のファイル(他のモジュール)からこのプロシージャを参照するにはプロシージャが宣言される前に，

```
public  start2          ←sample31.asm の中
```

のように宣言します．

また，参照する側のファイルでは，

```
extrn  start2:  near    ←sample31.asm の中
```

のように宣言します．

同じスコープ内では同じ名前を二重に宣言することはできません．「スコープ」とはラベルの通用範囲のことで，**PUBLIC**，**GLOBAL**の宣言がないときはそのファイルの中，そうでないときは宣言しているファイルと参照しているファイルです．

distance:　付けた方が望ましい．

NEARまたは**FAR**のどちらかを宣言します．デフォルト（何も付けないとき）は**NEAR**になります．プロシージャ内で使用される**RET**命令は自動的に，

RET	**C3**	**NEAR**	**IP**だけが**POP**される（**NEAR**）
RET	**CB**	**FAR**	**IP,CS**が**POP**される（**FAR**）

のどちらかを生成します．

呼び出す側（**JMP.CALL**）では**NEAR,FAR**などを付けないといけません．

USE item:　省略可，あまり使用されていない．

指定された**item**をプロシージャの入り口で**PUSH**し，出口で**POP**します．

argument :

Cなどの言語との整合をとるためのものです．この本では省略します．

RETURNS argument :

これもCなどの言語との整合をとるためのものです．この本では省略します．

ローカル・ラベル

アセンブラにTASMを使った場合，プロシージャ内ではプロシージャの外からは参照できないローカル・ラベルを使うことができます．

ローカル・ラベルには，

```
@@LABEL_NAME:
```

のように‘@’を2個ラベル名の先頭に付けます．

結論

PROC（プロシージャの宣言）は「サブルーチンの書き方」くらいに考えておいてください．

単純に以下のスタイルを覚えてください．

```
名前　PROC　NEAR
  ここにプログラム
名前　ENDP
```

このプロシージャが他のファイルから呼ばれるのであれば，このファイルの先頭またはプロシージャの直前で，

```
PUBLIC　　名前
```

と宣言します．

5 数式，定義のディレクティブ

EQUと「=」

EQUはシンボル（名前）に，エイリアス，式，文字列のいずれかを割り当てることができます．「=」は式だけを割り当てることができます．

したがって，式（数値）を割り当てるのであれば「=」を使うのが良いでしょう．予期しないトラブルを未然に防げるからです．

元のエイリアス，式，文字列の替わりにシンボルを使用できるようになります．

「**エイリアス**」はキーワードまたはニモニックです．これは再定義できません．「**式**」とは定数やアドレスです．

文字列，式中に文字列があるとき，それがエイリアス，定数，アドレスのいずれにも解釈されないときは文字列式として記憶されます．文字列は再定義できます．

例：

PC98テキストVRAMのセグメント定義

```
TVRAM_SEG  EQU    0a000h
TVRAM_SEG  =      0a000h
```

この２つは同じ意味です．

デバッグ用ルーチンの組み込み，取り外しのスイッチ

```
DEBUG      =      1          0に変えると組み込まれない．
IFIDN<DEBUG>,<1>
        デバッグ用ルーチン
ENDIF
```

6 ラベル関連

ラベルに関連した疑似命令には，**PUBLIC**，**EXTRN**，**GLOBAL**，**LABEL**，**LOCAL**などがあります．

ラベルは，

```
jmp    ラベル名
call   ラベル名
mov    ax,offset ラベル名
mov    ax,ラベル名
```

のように使います．実際のアドレスまたは数値はアセンブラが計算してくれます．

ソース・レベルで見ると同じ扱いなので，EQU，“=”で定義された変数も含めて「シンボル名」と呼ぶこともあります．

通常，ラベルを定義するとき，プロシージャ内では，

```
ラベル名:
```

とコロン(:)を付けます．

それ以外では，

```
ラベル名      db      '文字列'
ラベル名      dw      1
```

というようにコロン（:）は付けません．

また，プロシージャ名もラベルと同じように使用することができます．

```
jmp    プロシージャ名
call   プロシージャ名
```

PUBLIC

シンボルを他のモジュール（ファイル）から参照できるようにします．

使い方は，

```
PUBLIC      [ language ] symbol [ , [ language ]  symbol  ... ]
```

で，複数のシンボルを書くときはカンマ（,）で区切ります．

パブリック・シンボルとして宣言できるのは，

- □プロシージャ名
- □プログラムのラベル
- □データのラベル
- □**EQU**で宣言した数値定数

です．

［例］

```
PUBLIC      PROG1, PROG2, DATA1, CONST1
```

languageの部分は，「**NOLANGUAGE**」，「**C**」，「**PASCAL**」，「**BASIC**」などです．

この部分は今は必要ないので省略します．

必要なときはアセンブラ付属のマニュアルを見てください．

```
EXTRN
```

シンボルが他のモジュール（ファイル）で定義されていることをアセンブラに知らせます．

使用法は，

```
EXTRN       シンボルの記述  [, シンボルの記述  ... ]
```

です．

複数のシンボルを書くときは　コンマ(,)で区切ります．

シンボルの記述はつぎの形式です．

```
[ language ] name [ [ count1 ] ] : type [ :count2 ]
```

language　　省略可

NOLANGUAGE,C,PASCAL,BASIC,FORTORAN,PROLOGのいずれかを指定します．通常は使いません．

name　　省略不可

シンボル名です．

[count1]　　省略可

配列要素乗数です．通常は使いません．

: type　　省略しない方が良い．

シンボルがプログラムのときは**NEAR**，**FAR**，**PROC**のいずれか．**PROC**は**MODEL**疑似命令で指定されたモデルにより自動的に**NEAR**，**FAR**のどれかに設定されます．

プログラミング・スタイルにより一概にいえませんが，**NEAR**，**FAR**のどちらかを明示するのがよいでしょう．

シンボルがデータのときは，データのサイズにより**BYTE**，**WORD**，**DATAPTR**，**CODEPTR**，**DWORD**，**FWORD**，**PWORD**，**QWORD**，**TBYTE**または構造体名（**STRUC**で宣言される）を使います．

シンボルが宣言された数値のときは，ABSを使います．

:count2　　省略可

シンボルが**DUP**疑似命令で確保されているときその項目数を記述します．

［一般的な使用例］

```
EXTRN       LABEL1:NEAR , LABEL2:NEAR

EXTRN       DATA1:BYTE
EXTRN       DATA2:WORD
```

GLOBAL

GLOBALは**TASM**専用のディレクティブで，**PUBLIC**と**EXTRN**を組み合わせた機能をもちます．

シンボルが**GLOBAL**宣言と同じソース・ファイル内で定義されている場合は**PUBLIC**宣言となり，そうでないときは**EXTRN**宣言となります．

自動的に判別してくれるため，使うにあたっては便利です．また，ソース・ファイル間でプログラム（サブルーチン，プロシージャなど）を移動したときにアセンブル・エラーなどが出ることが少なく，良いのかも知れません．

逆に，ソース・ファイルからは外部のシンボルなのか内部のシンボルなのかが分かりづらい面もあります．

使用法は**EXTRN**と同じです．

LABEL命令

ラベル名　　**LABEL**　タイプ

目的：タイプ（サイズ）の違うデータをアクセスするために使用します．

注意：ラベルを宣言する前にこの宣言をします．

［使用例］

```
WORD_ACCESS     LABEL  WORD
BYTE_DATA  DB   32     DUP(0)

        MOV    WORD_ACCESS[2],1
```

とあった場合，

```
WORD_ACCESS     DW     16     DUP(0)
BYTE_DATA  DB   32     DUP(0)
```

の2つが重なって同じエリアにあると思えばよいのです．

LOCAL

目的：　①マクロ内でのみ使用できるラベルを定義する．
　　②プロシージャ内でのみ使える変数を定義する（TASMのみ）．
　　二重定義にならないようにするため使います．

マクロ内

LOCALはマクロ定義の最初の文として記述しなければなりません．マクロについては後述します．

LOCALで宣言したラベル（シンボル）はマクロ展開のたびに固有のシンボル名が割り当てられるため，二重定義になることはありません．

［文法］

```
LOCAL    symbol [, symbol ]
```

［使用例］

```
ifEquCall    MACRO Function
        LOCAL  None
        jne    Done          ;等しくなければマクロ終わり
        call   Function      ;等しければサブルーチンをコール
Done:
        ENDM
```

プロシージャ内（TASMのみ）

プロシージャ内ではスタック領域へアクセスするシンボル名が定義されます．

プロシージャ内で，そのシンボルが使われる前であれば，どこで宣言してもかまいません．

［文法］

```
LOCAL [[配列サイズ要素乗数]][:type][要素数][=サイズ・シンボル]
```

:type

BYTE，**WORD**，**DATAPTR**，**CODEPTR**，**DWORD**，**FWORD**，**PWORD**，**QWORD**，**TBYTE**，**NEAR**，**FAR**，**PROC**または構造体名が使われます．

省略すると**WORD**になります．

[配列サイズ要素乗数]

要素数

シンボルに割り当てられる領域のサイズ合計は，

```
[配列サイズ要素乗数] × 要素数 × type のサイズ
```

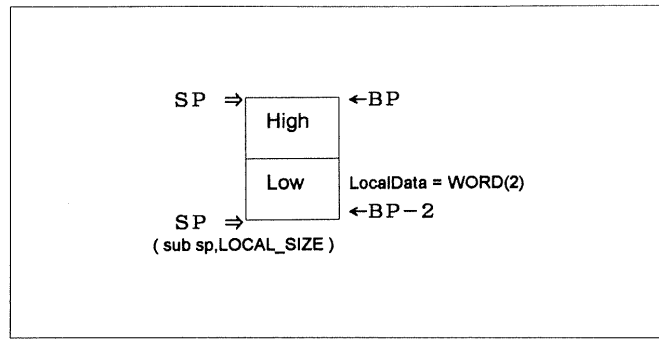
となります．

[サイズ・シンボル]

「シンボルに割り当てられる領域のサイズ合計」をシンボルを使って指定し，スタック領域内にローカル変数用の領域を確保することができます．

使用例：

```
Sample  PROC   NEAR
        LOCAL  LocalData:WORD=LOCAL_SIZE   ;LOCAL_SIZE=1*1*2
        push   bp                          ;になる
        mov    bp, sp
        sub    sp, LOCAL_SIZE ;            ローカル変数域を確保
        mov    LocalData, 0                ;LocalData=[bp-2]
        ;
        ;-- Sample の本体
        ;
        add    sp, LOCAL_SIZE ;            スタックを戻す
        pop    bp
        ret
Sample  ENDP
```



使用例からも推測できるように，主にC言語などの関数を作るときによく使われます．

7　マクロ

基本的には頻繁に使うプログラム数行をあらかじめ定義しておき，「マクロ名」または「マクロ名＋パラメータ」という形で使います．

マクロには，

①キータイプの回数を減らす．
②似たような数行を書くときのケアレス・ミスが防げる．
③論理的に分かりやすい記述にできる．

などの効果があるとされています．

プログラマーによっては「簡単なマクロしか使わない」，「複雑なマクロを書くことに喜びを感じる」など色々な人がいます．中には「ソースを分かりづらくするためにマクロを使う」人までいます．

マクロについて説明を始めると，それだけで1冊の本になってしまうので，ここでは基本的なことだけを述べます．

実働実績のあるマクロを蓄積していくと，それだけでもライブラリに匹敵するプログラマーの基礎能力の向上になるでしょう．

ただし，最初から大きなマクロを使おうとはせずにあまり無理をせず徐々に蓄積すればよいでしょう．

参考になるものは意外とたくさんあります．たとえば，C言語のスタートアップ・ルーチンのソース（たいがいのCコンパイラは付属または別売りしています），雑誌の特集，マクロ集の本などです．

しかし，他人の作ったマクロ集はなかなか使いこなせないものです．その第1の理由はマクロの名前です．他人の作ったマクロ名はなかなか憶えられませんし，思い出せません．そうなるとマクロの価値は半減してしまいます．ぜひともあなた自身のマクロ集を構築してください．もっとも，社内で共通のマクロを使うときなどはそうもいきませんが．

［文法（定義部分）］

```
Symbol      MACRO       [ parameter ] [, parameter ] ...

マクロ定義本体

ENDM
```

MACROの行で始まり，**ENDM**で終わります．

Symbol　　マクロ名です．この名前を使ってマクロを使います．

parameter　　パラメータです．パラメータなし，または1個以上のパラメータを使えます．

マクロ使用時のパラメータが定義時のパラメータに置き換えられます．

私見ですが，あまり多くのパラメータ（たとえば10個とか）は使うべきでないと思っています．

［使用例］

ファンクション09hを使用した文字列の出力マクロ例

```
定義部分
        BdosMsg         MACRO       Msg
                mov     dx,offset Msg
                mov     ah,09h
                int     21h
        ENDM

    使用（呼び出し）部分
      ....
        BdosMsg         StartMsg
      ....
```

SAMPLE2.ASMのメッセージ出力部分をマクロ化したのが**SAMPLE22.ASM**です．

```
A>exe sample22
```

でアセンブルして確認してみましょう．

SAMPLE22.LSTの**BdosMsg**の部分を見てみましょう．

```
                                        ↓マクロ呼び出し
18                                      BdosMsg AsmMsg
19      0005    BA 0000r    mov         dx,offset AsmMsg    ↓展開された部分
20      0008    B4 09       mov         ah,09h
21      000A    CD 21       int         21h
22
```

ソース中で3行書くより，1行だけですむ方が簡単だし，ケアレス・ミスも起こりにくいのは容易に想像できます．また，ソースも見やすくなりメンテナンスも楽になるでしょう．

マクロ中で使える条件ディレクティブには多くのものがあります．それらを使って非常に複雑なマクロも実現可能ですが，あくまで使いやすいマクロを作ることを心がけてください．

マクロ中のみで使えるif文

IFB　<Parm>	パラメータが与えられないと真
IFDIF <Parm1>,<Parm2>	２つのパラメータが異なると真
IFIDN<Parm1>,<Parm2>	２つのパラメータが同じだと真
IFNB　<Parm>	パラメータが与えられると真
ENDIF	if文の終わり

詳細はアセンブラのマニュアルを見てください．

8　°インクルード・ファイル

インクルード・ファイルは共通の定義などを1か所（1本のファイル）で管理し，メンテナンスなどの効率を良くするために使います．

共通の定義とは機種依存の部分（テキスト画面のアドレス，プリンタのポート，特定のコマンド・コードなど），ASCIIコードの定義，マクロ定義，開発プログラム特有の定義（データのサイズ，コマンド番号，バッファのサイズなど）があります．

ある程度以上大きなプログラムを開発するときは複数のソース・ファイルに分割し開発するのが一般的ですし，また，そうしないとエディタ，アセンブラなどの能力からも効率が悪くなったり，開発そのものが不可能となってしまいます．

どのようにソース・ファイルを分割するかはプログラマーによって意見も分かれるでしょうし，個性の現れるところでもありますが，機能によって分割するのが一般的です．

ざっと考えても次のような分類が思いつきます.

```
初期化部分
キーボード
マウス
コマンド解析部分

ファイルの読み書き
画面表示
プリンタ

EMSの処理

データ処理部分
```

特定部分の機能が肥大すればさらに分割する必要があるでしょう.

各ファイルで同じ定義をそれぞれしていたのでは，ファイルによって違う値を定義してしまうミスをしやすいでしょう．当然，修正するのも大変です.

たとえば，キーボードから文字を入力し，それに従って処理をするプログラムを考えてみましょう.

```
キーボード          押されたキーにより文字を取得
↓    文字
コマンド解析部分    有効な文字ならコマンド番号に変換
↓    コマンド番号
処理部分            コマンド番号により処理をする
```

コマンド番号は「コマンド解析部分」と「処理部分」で共通して使います.

```
COM_END       EQU     0
COM_READ      EQU     1
COM_WRITE     EQU     2
COM_DISPLAY   EQU     3
```

などのように定義したものを使うと良いでしょう.

定義ファイルのファイル名を「**COMMAND.NO**」とすると，「コマンド解析部分」と「処理部分」のファイルの先頭に次のように記述します.

```
INCLUDE     COMMAND.NO
```

「キーボード」と「コマンド解析部分」では文字コードを共通して使っています．アルファベットと数字のみを使うのであれば,

```
'A' 'Z' '1'
```

というように記述しても読みづらくありませんし，プログラムを書くのも大変ではありません.

しかし，CTRL-Cやファンクション・キーなどの入力も処理するようになると，

```
cmp   ax,03h          ; 03h = Ctrl-C
cmp   ax,1ah          ; 1ah = Ctrl-Z
```

などと書くのは間違いを起こしそうですね．それならば，

```
CTRL_A          EQU   01h
CTRL_B          EQU   02h
CTRL_C          EQU   03h
CTRL_D          EQU   04h
CTRL_E          EQU   05h
CTRL_F          EQU   06h
.....
```

というような定義を「**KEYCODE.DEF**」というファイルに定義して，「キーボード」と「コマンド解析部分」のソース・ファイルに，

```
INCLUDE     KEYCODE.DEF
```

と記述します．

```
cmp   ax, CTRL_C
cmp   ax, CTRL_Z
```

というように使えます．

INCLUDEの文法：

```
INCLUDE     ファイル名
```

ファイル名にはドライブ，ディレクトリ名も記述できます．

したがって，上記の「**KEYCODE.DEF**」などのプログラム開発で共通に使う定義ファイルは，たとえばサブディレクトリ**<ASM_INC>**を作って，その中に置いておくと良いでしょう．

```
¥-┐
 ├-<TASM>
 ├-<ASM_INC>          ←例えばこのように
 ├-<WORK1>
 ├-<WORK2>
 └-<WORK3>
```

次のように使います．

```
INCLUDE     ¥ASM_INC¥KEYCODE.DEF
```

IN JCXZ
HF LDS LEA
CK LODS LOOP
OP NOT OUT
SH PUSHF REP
EG STOS WAIT

他人のプログラムを大いに参考にしよう

第4部　逆アセンブラ『ZZ386』を使う

ここまで読み終えたあなたの今後の課題

ここまで来るとあなたはもうプログラムが書けるはずです．
とにかく書いてみましょう．プログラムを創る体験をしないと先には進めません．
しかし，すべてを自分自身で創造していたのでは人生が終わってしまいます．
8086が世に出てから10年余りが過ぎましたが，全世界で何十万というアセンブラ・プログラマーの創造を追体験しようとしても，とても不可能なことです．

XCHG XLAT
DD CMP CMPS
BB SCAS SUB
OR SAR SHL
TEST XOR

1　プログラム上達の近道

何であれ上達するには教科書，教材が必要です（いちばん重要なのは頭を使うことですが）．

しかし，プログラム，特にアセンブラやC言語に関しては良い教科書，良い教材は多くありません．

優秀なプログラマーであれば教科書を書くよりプログラムを作るほうが，教材を作るより実用的なプログラムを開発するほうが，経済的な見返りも地位や名誉も得られるというものです（??）．

（少なくとも日本では…海外の状況は私は知りません）

したがって，

実用的なプログラムこそ知識の宝庫

なのです．

しかし，そのようなプログラムは教材として開発されたわけではないので，そのままでは内容を見ることはできません．そのために「逆アセンブラ」というツールが非常に有効です．

最近は「Vzエディタ」のようにソース・プログラム付きで販売されているソフトもありますが，残念ながら今後ともそのようなソフトの比率が増えるとは思えません．

また，それらのソフトも教材として開発されたのではありませんので，ソース・プログラムがあれば何でもできるというものではありません．

私はVzエディタの作者の兵藤氏には敬意を3つぐらい払っています．

世の中にたくさんある教材の中から効率良く知識を吸収しましょう．

これは「著作権を無視しよう」といってるのではまったくありません．

「コピー」と「知識を得る」ことはまったく違います．

市販されているソフトウェアの中にはリバース・エンジニアリング（いわゆる逆アセンブル）の禁止を使用条件に折り込んだものもあります．そういう条件を付けることの是非についてはとりあえずおいておいて，そのような使用契約（使用条件）は法律的には有効であるというのが一般的な考えのようです．

（このことに関して意見は色々あるでしょうが，私は反対です．なぜなら彼ら自身も先人のプログラムから多くのことを学んできたはずだからです．）

したがって，そのような使用条件の付いた市販ソフトを逆アセンブルすることはおすすめしません．

ここの結論：どんどん逆アセンブルしましょう．

2 『ZZ386ミニ』の仕様

エディタ風の操作性のよい逆アセンブラ『ZZ386』を体験し，逆アセンブラの有用性を実感してもらうために，本書には体験版『ZZ386ミニ』が付属されています．『ZZ386ミニ』は市販されている『ZZ386』と比較して次の機能的制限があります．以下『ZZ386ミニ』を「ミニ版」と呼びます．

PC-9801版のみ

「製品版」のパッケージには，PC-9801版以外に，IBM PC/ATで動作する英語版とDOS/Vで動作する日本語版が標準で含まれています．

「ミニ版」はPC-9801シリーズでのみ動作します．

CPUは8086のみ

「製品版」では8086～80486のすべての命令と，数値演算プロセッサ命令すべてをサポートしていますが，「ミニ版」は8086の命令のみサポートしています．また数値演算プロセッサの命令はサポートしていません．

逆アセンブルできるファイルはCOM，EXEのみ

「製品版」ではCOM，EXE，デバイスドライバ（SYS，DRVなど），DOSエクステンダの実行ファイルEXP，オーバーレイ・プログラム，Windows用プログラムが逆アセンブルできます．

「ミニ版」ではCOM，EXEのみ逆アセンブルできます．

ターゲット・ファイルは2KB以下

「ミニ版」ではロード・モジュールのサイズが2Kバイト以下の実行ファイルが逆アセンブルできます．

EMS，ディスク・スワップなし

「ミニ版」ではEMSおよびXMSは使えません．また，データ処理でディスク・スワップはしていません．

動作環境の違い

「ミニ版」は実行ファイル，データ・ファイルなどを同じディレクトリに入れて実行するようになっています．

「製品版」は実行ファイル，作業ディレクトリを別のディレクトリに置き，通常のMS-DOSコマンドのような環境で使います．

3　『ZZ386』の使い方の概略

使い方は簡単です．そんなに難しく考えないでください．
まず，必要なファイルをコピーして，（インストール）
何を逆アセンブルするか決めます．（プロジェクト・ファイル）

実際に『ZZ386ミニ』または『ZZ386』を動作させながら確認してください．

『ZZ386ミニ版』のインストール

製品版では専用のインストール・プログラムが付いていますが，ミニ版では簡単なバッチ・ファイルになっています．

このバッチ・ファイルは『ZZ386ミニ』をインストールしたいドライブの“**ZZ386**”というサブディレクトリにファイルをコピーします．

サンプル版はハードディスクで使うことを前提として作ってありますが，フロッピーディスクでも使えます．

ハードディスクを「**A:**」とし，付属のフロッピーディスクを入れたドライブを「**B:**」とします．

次のようにバッチ・ファイルを実行するとAドライブに「ZZ386」というディレクトリを作り，そこに『ZZ386ミニ』をコピーします．

```
A>B:INSTZZ B A
```

ミニ版を実行するときは，カレント・ディレクトリをこのディレクトリに移動してから実行してください．

プロジェクト・ファイルの作成

「ZZ386」を使って逆アセンブルをするには，

ターゲット・ファイル
Target File

①ターゲット・ファイル（逆アセンブルするファイル）
②ファイルのタイプ（COM，EXE，SYSなど）
③CPUの種類（8086，80286，80386など）
④拡張メモリ使用の指定（EMS）

など多くの指示が必要です．

「ZZ386」を起動するたびにそれらの指示をするのは煩雑でもあり，トラブルの原因にもなります．

プロジェクト
Project

「ZZ386」では1つのターゲットの逆アセンブル（プロジェクト）ごとに，それらの設定をファイル（**プロジェクト・ファイル**）に記録しておきます．

ターゲットの解析を始めるにあたっては，まずこのプロジェクト・ファイルを作成します．

新しくプロジェクト・ファイルを作成

新規にプロジェクト・ファイルを作成するには，

コマンド・ラインからプロジェクト名を指定する．

A>ZZ386 プロジェクト名.**ZPJ**

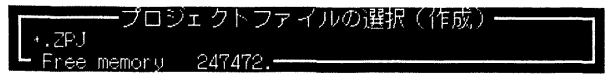
とタイプする．

プロジェクト・ファイル名入力ウィンドウから入力する．

A>ZZ386

とタイプして，ZZ386を起動します．

最初にプロジェクト・ファイル設定の状態になります．



この状態で，新規のプロジェクト・ファイル名をタイプします．

このように指定されたプロジェクト・ファイルが存在しないと（新規のファイルだと）自動的にプロジェクト内容設定モードに画面は切り変わります．

参照：プロジェクト・ファイルの諸設定

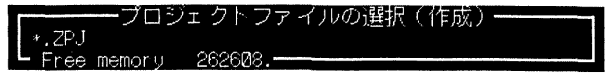
プロジェクト・ファイルを選択

既存のプロジェクト・ファイルを選択し，逆アセンブルを再開するときは，前項の「新しくプロジェクト・ファイルを作る」と同様に，

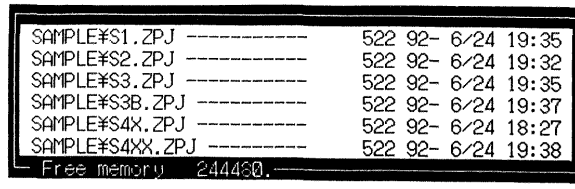
①コマンドラインから
②プロジェクト・ファイル名入力ウィンドウから

のいずれかから選択できます．

次の状態（プロジェクト・ファイル名入力ウィンドウ）から，



が表示されているときに RETURN を押すとプロジェクト・ファイルが一覧表示されます．



この中から簡単に選択できます．

プロジェクト・ファイルの設定変更

プロジェクト・ファイルの内容の変更（環境，条件の変更）は，

ZZ386起動時に

ZZ386実行中はオプション・メニューから

のいずれかから変更できます．

ZZ386起動時に

後述するように起動後も設定は変更できますが，「ターゲットCPU」，「拡張メモリの使用」などの項目を変更したときは，ZZ386を再起動する必要があります．

そのような変更が必要なときは起動時に変更した方が便利です．

A>ZZ386 /P

または，

A>ZZ386 プロジェクト・ファイル名 **/P**

でZZ386を起動します．

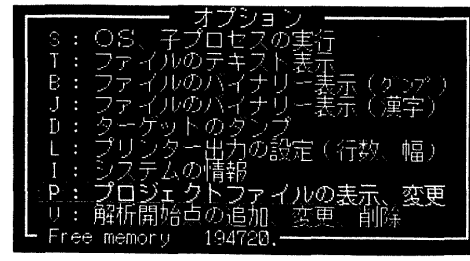
プロジェクト・ファイルが選択されると自動的に「プロジェクト・ファイル変更モード」になります（プロジェクト・ファイルの諸設定を参照）．

オプション・メニューから（ZZ386実行中）

O または f-2 を押すとオプション・メニューが表示されます．

P を押すか，カーソルを移動して「プロジェクト・ファイルの表示，変更」を選択します．

プロジェクト・ファイルの設定内容が表示されます（プロジェクト・ファイルの諸設定を参照）．



プロジェクト・ファイルの諸設定

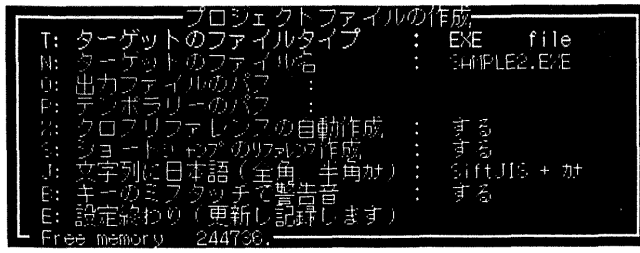
プロジェクト・ファイルには逆アセンブルの条件，環境などの設定が記録されています．

設定内容はメニュー形式で表示されます．

各項目を選択し変更します．

項目は1文字のイニシャルのキーかカーソル移動と RETURN で選択します．

以下の説明のうち，「‡」のついた項目はミニ版では設定できません．



ターゲットCPU‡

ターゲットCPU（ニモニック）を選択します.

8086～80486および数値演算プロセッサの命令が選択できます.

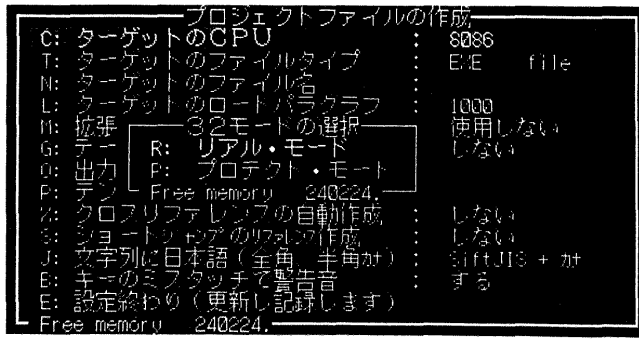
メニューで一覧表示されます.

NDPと記されているものは数値演算プロセッサの命令を含んでいます.

80486を選択すると自動的に数値演算プロセッサの命令が含まれます.

EXTRAはニモニックテーブル・ファイルを作成し，標準以外のCPUに対応したときに選択します.

80386以上のCPUを選択したときはベース・モード選択のメニューが現れます.



「ミニ版」は8086以外のCPUの命令は逆アセンブルできないので，この項目はメニューに表示されません.

ターゲットのファイルタイプ

ZZ386は様々な形式のファイルを逆アセンブルできます.

メニューから選択します.

□COMファイル

拡張子が“**.COM**”の実行ファイルを選択します.

□EXEファイル

拡張子が“**.EXE**”の実行ファイルを選択します.

□BINファイル‡

ROM化されたプログラムやメモリ上のプログラムをファイルに落としたものを解析します.

このタイプを選択すると実行開始点を指定する必要があります.

デバイス・ドライバ
Device Driver

□デバイスドライバ‡

デバイスドライバは特殊なファイル形式です.

拡張子が“.SYS”, “.DRV”などのファイルがこれにあたります.

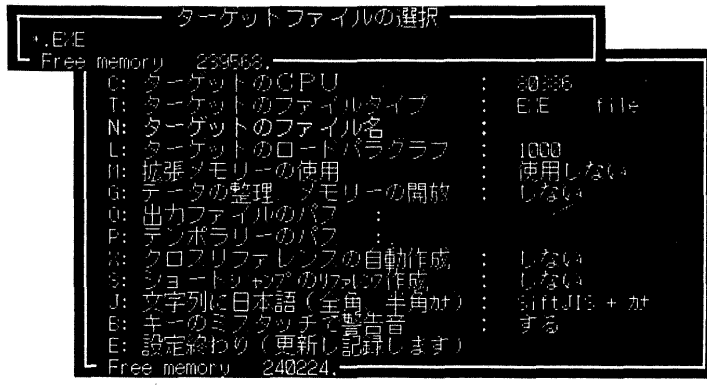
□EXPファイル‡

DOSエクステンダ用の実行ファイルです.

拡張子は“.EXP”となっています.

ターゲットのファイル名

選択されたファイル・タイプにより，それぞれデフォルトの拡張子がセットされます．たとえば，EXEファイルを選択したときは，

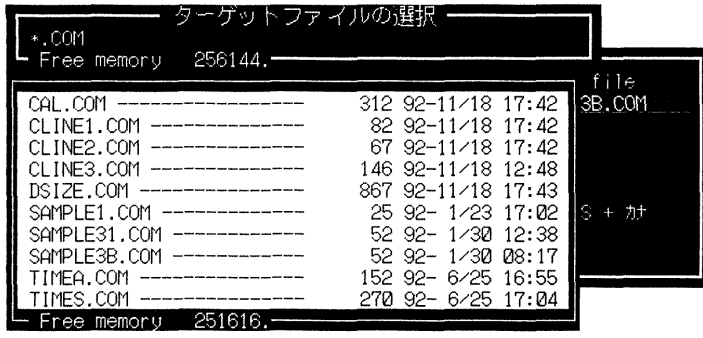


と表示されます．

拡張子が違っていれば変更します．

フルネームを入力しても構いません．

このまま**RETURN**を押すと“***.EXE**”のファイルが一覧表示されます．



逆アセンブルしたいファイルにカーソルを合わせて**RETURN**で選択します．

ターゲットのロードパラグラフ‡

ロード・パラグラフ
Load Paragraph

通常，MD-DOSの実行ファイルは，メモリの不特定の部分に読み込まれて実行されます．当然，実行ファイルはメモリのどの位置からでも実行できる構成になっています．

したがって，プログラムは低位番地（割り込みベクタやシステム・データがある）や640Kを越える高位番地（システムROMやビデオRAMなどがある）に割り付けないかぎり，解析の支障になることはありません．

デフォルトでは，解析上判別しやすいようにプログラムはアドレス「**1000:0000**」から置かれるようになっています．

一方，ROM化されたプログラムやメモリからファイルに落とされたプログラムはリロケートの情報をもっていないので，ターゲットと同じアドレスに設定しないと正しく解析されません．

セグメント
Segment

パラグラフ（セグメント）を指定します．

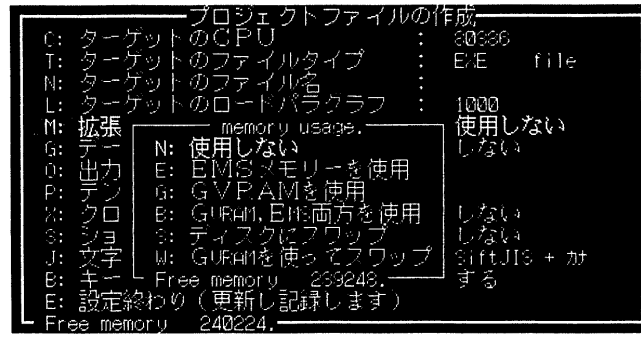
拡張メモリの使用‡

GVRAM
システム・メモリの領域の中にあるグラフィック・データを格納する領域．

EMS，GVRAM，ディスク・スワップを使って，より大きなファイルの解析，効率の良い解析ができます．

ディスク・スワップ
Disk Swap

メニューから選択します．



□使用しない

コンベンショナル・メモリ
Conventional Memory

通常のメモリ（コンベンショナル・メモリ）だけを使います．
（ミニ版はこの設定になっています）

□EMSメモリを使用

EMS＋コンベンショナル・メモリを使います．

□GVRAMを使用

GVRAM＋コンベンショナル・メモリを使います．

□GVRAM，EMS両方を使用

EMS＋GVRAM＋コンベンショナル・メモリを使います．

□ディスクにスワップ

コンベンショナル・メモリをバッファに使い，データをディスクにスワップします．

□GVRAMを使ってスワップ

GVRAMをバッファに使い，データをディスクにスワップします．

データの整理，メモリーの開放‡

各処理フェイズごとに，スワップ，解放，整理をしてメモリ効率を良くします．スピードはその分遅くなります．

フェイズ
Phase

出力ファイルのパス

逆アセンブルで得られるソース・ファイル，解析結果のデータなどを出力するパスを指定します．

通常はカレント・ディレクトリに出力されます．

カレント
Current

テンポラリーのパス

テンポラリ
Temporary

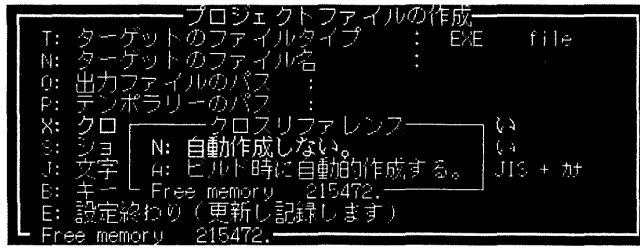
ZZ386は自動解析中に，メモリ使用効率をよくするため一時データをディスクに記録しています（禁止できません）．

RAMディスクなどを指定すると自動解析のスピードが速くなります．

クロスリファレンスの自動作成

クロス・リファレンス
Cross Reference

自動解析を実行したときにクロス・リファレンスを自動作成する設定です．



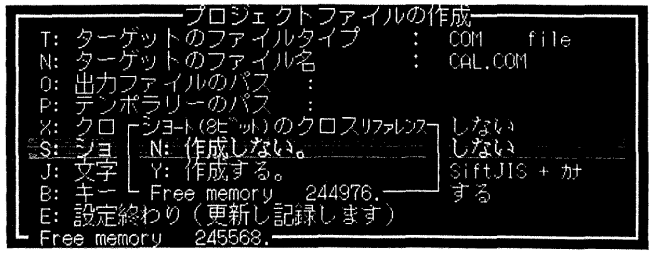
デフォルトでは自動作成はおこなわれません．

ショートジャンプのリファレンス作成

クロス・リファレンスのデータは非常に多くのメモリを消費します．

ショート・ジャンプ（SHORT，条件ジャンプ）は，情報としてはそれほど有効ではありません．これらのデータを作らないことでメモリを節約します．

ジャンプ
Jump

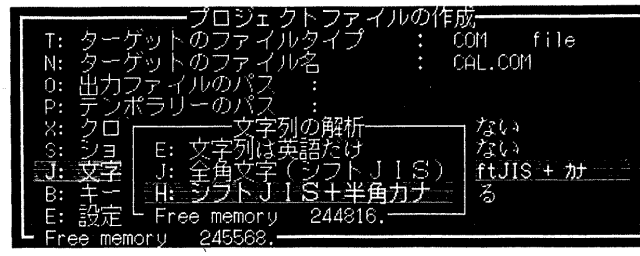


デフォルトでは作られません．

文字列に日本語（全角，半角カナ）

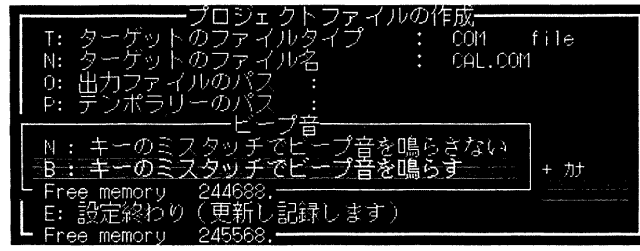
文字列データの解析での日本語処理を指定します．

メニューよりいずれかを選択します．



キーのミスタッチで警告音

無効なコマンド（キー）などを入力したときピッと音を出しますが，これを出さなくします．



設定終わり（更新し記録します）

変更されたデータは「設定終わり」を実行しないと記録されません．

逆アセンブルの実行

「ZZ386」を起動すると，最初にプロジェクト・ファイルが読み込まれます．

プロジェクト・ファイルの設定に従ってニーモニックのファイル，解析の対照となるターゲットのファイルが読み込まれます．

解析の再開であれば解析結果の種々のデータファイルが読み込まれます．

1回目の解析であれば，プログラムの実行開始点から単純にコードと見なして1ページ分表示されます．

「逆アセンブル」と一口にいっても，ターゲットの大きさ，種類など色々なケースがあります．またその目的により色々な苦労もあります．まずは簡単なところから…．

ターゲットのプログラムがアセンブラで書かれた実行ファイルのときは，比較的解析しやすい構造となっています．

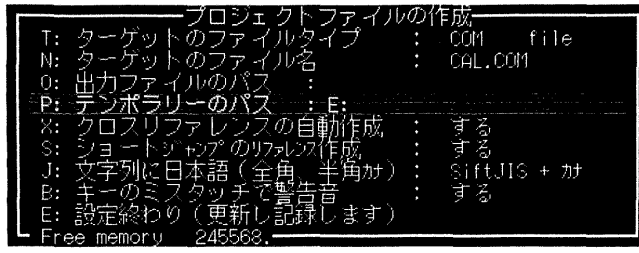
ZZ386の諸設定をデフォルトのままで実行しても，ほぼ完全なソース・プログラムを得ることができます．

例として，「**CAL.COM**」を逆アセンブルしてみましょう．

プロジェクトファイル名を「**CA.ZPJ**」として「ZZ386」を起動してみましょう．

```
A>ZZ386 CA.ZPJ
```

最初にプロジェクトファイル設定の画面が現れますので，次の設定にしてください．



この中で，

```
P: テンポラリーのパス　: E:
```

の部分はあなたの環境に合わせてください．

設定が終わったら，必ず「設定終わり」を実行します．

```
E: 設定終わり（更新し記録します）
```

にカーソルを合わせ**RETURN**を押すか，**E**を押します．

ニモニックのファイル，ターゲットのファイル「**CAL.COM**」が読み込まれると，実行開始点「1000:0100」から1ページ分が表示されます．

最上行にはタイトル，プロジェクト・ファイル名，ターゲットCPUが表示されます．

最下行にはコマンドのガイド（ファンクション・キーとショートカット・キー）が表示されます．

2行目から24行には逆アセンブルされたプログラムが表示されています．

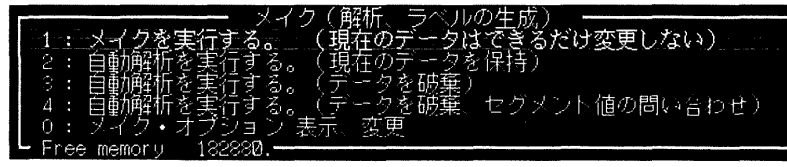
このときはまだ解析が行なわれていないので，ラベルなどは表示されません．

メイク（自動解析）

逆アセンブルの実行，つまりプログラムを解析してラベルやプロシージャを作成したり，セグメントの区分を自動的に行なうことを「**メイク**」と呼んでいます．

最下行にはコマンドのガイドに「Make」と表示されています．
これはメイク・コマンドは[f･10]または[M]で実行されることを表わしています．

[f･10]（または[M]）を押してみましょう．次のメニューが現われます．



ここでは標準的なモードの，

3：自動解析を実行する。（データを破棄）

を選択するので，[3]を押します．

数秒で解析は終わり，その後自動的にクロス・リファレンスのデータが作られます．

両方で10秒程度でしょう．使用するパソコン，テンポラリのパスにRAMディスクを使えばもっと速くなるでしょう．

クロス・リファレンス作成中は次の表示が現われ，順次データ数が表示されます．

クロス・リファレンスの作成が終了すると，次のメッセージが表示されるので何かキーを押します．

解析で得られたラベル，プロシージャ，コメントなどの情報とともに逆アセンブルされたソースが表示されます．

エディタと同様のキー操作でスクロール，ページ送り，カーソル移動してプログラムの任意の場所を眺めることができます．

また「ジャンプ」，「サーチ」などエディタにはない機能を使って効率よくプログラムを見ることができます．これらについては「ZZ386のコマンド」で詳しく述べます．

逆アセンブル結果の出力

とりあえず逆アセンブルし得られたプログラムを出力してみましょう．

逆アセンブル結果の出力は，f·9 または U で実行されます．

ファイル形式のオプション

最初にメニューから「オプション（リストの形式）」を選択します．

出力ファイルにアドレス情報なども記載されていたほうが便利なこともありますが，アセンブルするには不用です（エラーになります）．

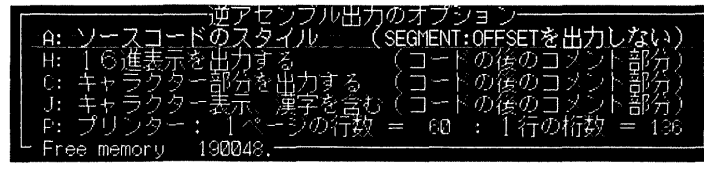
オプションを次の設定にします．

カーソルを合わせ RETURN を押すと現在の設定が反転します．

16進表示を出力する
キャラクター部分を出力する

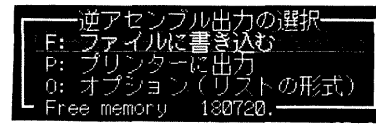
は「しない」に設定すると出力ファイルのサイズが小さくなります．ディスクの空き容量に不安があるときに設定します．

ESC キーを押してオプション・メニューを終了します．



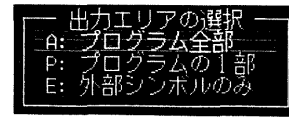
出力デバイスの選択

「ファイルに書き込む」を選択します．



エリアの指定

「プログラム全部」を選択します．

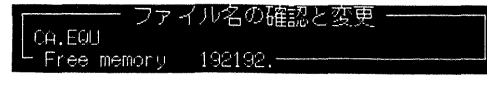


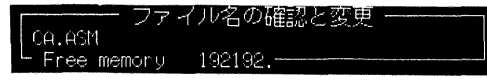
出力ファイル名の確認

ソース・プログラムとインクルード・ファイルの2つのファイルが出力されます．

ファイル名はプロジェクト・ファイル名（**CA.ZPJ**）にしたがい，それぞれ「**CA.ASM**」，「**CA.EQU**」になります．

確認のため表示されます．別の名前で出力したいのであれば，このときに新しいファイル名を入力します．





ZZ386の終了

ZZ386を終了するには f·1 または E を押します.

メニューが表示されるのでいずれかを選択します.

「データをセーブして終了」を選択すると解析で得られたデータをすべてファイルに記録するので次回の解析はその時点から始められます.

アセンブルの実行（プログラムの再現）

逆アセンブルの精度を確かめる1つの方法は，逆アセンブルで得られたソース・プログラムをアセンブルして実行ファイルを得ることです.

①アセンブルがエラーなくできるか？
②作られた実行ファイルが元の実行ファイルと等しいか？

逆アセンブルによって「**CA.ASM**」と「**CA.EQU**」が生成されているので，アセンブル，リンクをしてみましょう．TASMを使う場合，

```
A>TASM CA
A>TLINK /t CA,CA
```

で「**CA.COM**」が作られます.

この「**CA.COM**」を元のプログラム「**CAL.COM**」と比較してみましょう.

```
A>FC CAL.COM CA.COM
ファイルの比較中です. CAL.COM と CA.COM
FC: 違いは見つかりません.
```

と表示されるでしょう.

4　「ZZ386」のコマンド

ZZ386の操作は大きく3つに分類できます．

①カーソル移動，スクロールなどの画面操作．
②ファンクション・キーに割り当てられている主なコマンド
③属性の変更などアルファベット・キーだけに割り当てられている簡単なコマンド

選択肢のあるコマンドはメニューが表示され，その中で選択する構造になっています．

メニューでは，

①カーソル移動と RETURN
②メニュー内に表示される1文字のショートカット・キー（英数字）
③ファンクション・キーのガイドが表示されているときはファンクション・キー

のいずれかで選択，決定できます．

コマンドのキャンセルは ESC キーです．

マニュアルレス
Manual-less

全体に統一されたコマンド操作体系になっているのでマニュアルレスで操作できます．

オンライン・ヘルプ
On-line Help

また，オンライン・ヘルプ機能が充実しているので，操作が分からないときはとりあえず HELP キーを押せば，そのとき実行しようとしているコマンドのヘルプが表示されます．

カーソル移動，スクロールなどの画面操作

ヘルプの表示

操作方法が分からないときは HELP を押してください．

ヘルプ・メニュー
Help Menu

ヘルプ・メニューが表示されます．

各コマンドのメニューが表示されている状態では，コマンドに対応したヘルプが表示されます．

キーワード
Keywords

ヘルプの文中のキーワードを選択すると，キーワードに関するヘルプが表示されます．

ROLL UP，ROLL DOWN で前後のヘルプが表示されます．

ヒストリ
History

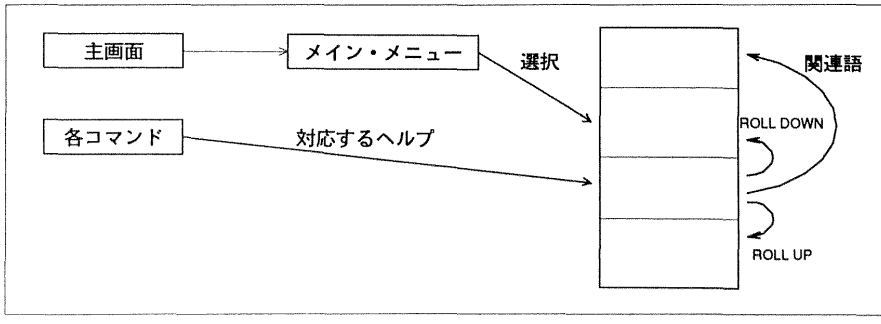
表示されたヘルプはヒストリとして残り，SPACE キーを押すと1回前のヘルプが表示されます．

＊ヘルプ本文の例

```
                ヘルプの項目を選択して下さい。
                カーソル移動、スクロール、キャラクタ表示
                メイン画面からのコマンド一覧（ｆキー）
                メイン画面からのコマンド一覧（ｆキー以外）
                属性の変更　（カーソル行のメイク）
                属性の変更　（部分的なメイク）
                プロシジャーの変更                              P
                ＳＲＥＧ、メモリーのロード値の表示、変更        U
                セグメント値の表示、非表示                      [ ]
  [ ｆ　１ ]    ＺＺ３８６の終了                ( Ｅ＋ )
  [ ｆｆ　２ ]  オプション                      ( Ｏ＋ )
  [ ｆｆ　３ ]  コメントのセット、削除          ( Ｃ＋ )
  [ ｆｆ　４ ]  ラベルのセット、削除            ( Ｌ＋ )
  [ ｆｆ　５ ]  ジャンプ                        ( Ｊ＋ )
  [ ｆｆ　６ ]  サーチ                          ( Ｓ＋ )
  [ ｆｆ　７ ]  セクメント値の変更              ( Ｇ＋ )
  [ ｆｆ　８ ]  クロスリファレンス              ( Ｘ＋ )
  [ ｆｆ　９ ]  逆アセンブルの出力              ( Ｕ＋ )
  [ ｆ　１０ ]  逆アセンブルの出力　ファイル、プリンター
                メイク　逆アセンブルの実行      ( Ｍ＋ )
                Ｄｏｓ割込み
                ｉｎｔ２１　　ページ　１／５
  [ｓｆ１０ ]   画面の修復
                現在のバージョンの制限事項              (1/3)
 ↑↓　で移動。　　[RETURN]で項目選択。　　[ESC]でキャンセル。
```

```
                プロジェクトファイルの設定、変更　　（ Ｏ＋Ｐ ）

ＺＺ３８６ではターゲットの様々な指定をプロジェクトファイルを作成して行います。
プロジェクトファイルの作成、変更はつぎのときに行われます。
  □ 新しいプロジェクトファイルを指定したとき。
  □ /P オプションを指定しＺＺ３８６を起動したとき。
  □ Ｏ+Ｐ でプロジェクトファイルコマントを呼びたしたとき。

        セットする項目にはつぎのものがあります。
        「 Ｅ 設定終わり（更新し記録します）」を実行しないと記録されません。

   Ｃ  ターゲットのＣＰＵ                  ：  80386
   Ｔ  ターゲットのファイルタイプ          ：  EXE FILE
   Ｎ  ターゲットのファイル名              ：  filename.exe
   Ｌ  ターゲットのロードパラグラフ        ：  1000
   Ｍ  拡張メモリーの使用                  ：  EMS
   Ｇ  データの整理、メモリーの開放        ：  しない
   Ｏ  出力ファイルのパス　：
   Ｐ  テンポラリーのパス　：D:
   Ｘ  クロスリファレンスの自動作成        ：  しない
   Ｓ  ショートジャンプのリファレンス作成  ：  しない
   Ｊ  文字列に日本語（全角文字）          ：  する
   Ｅ  設定終わり（更新し記録します）

←→ [RETURN]で関連語選択　[SPACE] 前の表示　[R-DOWN][R-UP]前.次頁　ESC キャンセル
```

＊ヘルプシステムの図



コマンドのキャンセル

コマンドのキャンセル，メニューからの復帰，メイクの中断などは ESC キーです．

カーソル移動とスクロール

カーソル移動は一般的なエディタと同じ操作になっています．

《カーソルのアップ，ダウン》

↑ キーまたは Ctrl - E でカーソルは1行上に移動します．

↓ キーまたは Ctrl - X でカーソルは1行上に移動します．

カーソルを右，左

オペ・コード
Operation Code

「ラベルのセット，削除」などでは，カーソルがオペコード側，オペランド側いずれにあるかによって処理が違います．

```
        処理されるラベル              処理されるラベル
               ↓                            ↓
C1000:01D6 CI_101D6: MOV    SI,Offset ES 10853      ;BE 0853

                       ↑                ↑
カーソル：         オペコード側        オペランド側
```

→ キーまたは Ctrl - D でカーソルはオペランド側に移動します．

← キーまたは Ctrl - S でカーソルはオペコード側に移動します．

《スクロール・アップ，ダウン》

Ctrl - Z　1行上にスクロールします．

Ctrl - W　1行下にスクロールします．

《ロール・アップ，ダウン》

ROLL UP または Ctrl - R で前ページが表示されます．

ROLL DOWN または Ctrl - C で次ページが表示されます．

ファンクション・キーに割り当てられている主なコマンド

主なコマンドはファンクション・キーとアルファベット1文字に割り当てられています.

画面の最下行にガイドが表示されています.

左からf·1, f·2 .... と表示されています.

大文字（強調色）で表示されているのが対応するアルファベット1文字のコマンド・キーです.

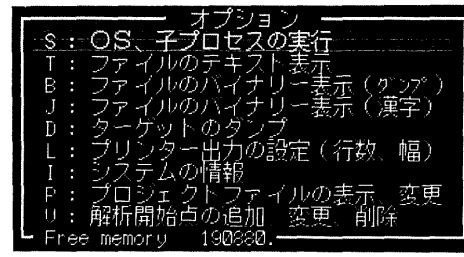
「ZZ386」の終了

f·1 または E を押すと終了メニューが表示されます.

解析で得られたラベル，プロシージャ，セグメントなどをデータを記録するがどうかを選択します.

オプション・メニュー

f·2 または O を押すとオプションのメニューが表示されます．



ＯＳ，子プロセスの実行

「ZZ386」を中断して，MS-DOSの「**COMMAND.COM**」（正確には「**COMSPEC=**」で指定されたもの）に制御を渡します．

コマンド・ラインからEXITとタイプすると「ZZ386」に戻ります．

「ZZ386」が300kバイト前後のメモリを占有していますので，それほど大きなプログラムは実行できません．

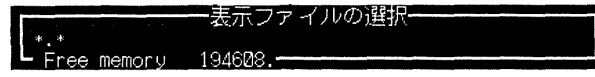
子プロセス中は識別のためプロンプトが変わります．

```
[ZZ386]A>
```

ファイルのテキスト表示

実行中にソース・ファイル，ニモニック・ファイル，メモなどを確認したくなることがよくあります．そのようなときに使います．

最初にワイルド・カードを含むパスが表示されます．



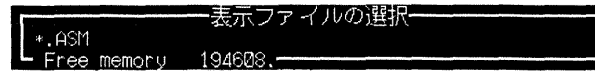
アセンブラのソース（**.ASM**）を見たいのであれば，次のように変更し RETURN を押します．

フル・パス
Full Path

もちろん，ファイル名はフル・パスで書いてもかまいません．

パス
Path

ここで指定したパスは記憶されていて次回からは，そのパスが表示されます．



パスに当てはまるファイルが一覧表示されます．この中から表示するファイルを選択します．

```
CA.ASM ------------------      4638 93- 1/26 14:56
CAL.ASM -----------------      2895 92-11/18 13:22
CLINE1.ASM --------------      2020 92-11/18 12:07
CLINE2.ASM --------------      1317 92-11/18 12:47
CLINE3.ASM --------------      2296 92-11/18 12:47
DSIZE.ASM ---------------      7232 92-11/18 17:26
FS.ASM ------------------     12651 92-11/18 16:40
SAMPLE1.ASM -------------       276 92- 6/25 18:00
SAMPLE2.ASM -------------       459 92- 7/ 3 18:01
SAMPLE22.ASM ------------       628 92- 7/ 3 18:01
SAMPLE31.ASM ------------       357 92- 7/ 3 18:01
SAMPLE32.ASM ------------       317 92- 7/ 3 18:01
SAMPLE41.ASM ------------       426 92- 7/ 3 18:01
SAMPLE42.ASM ------------       337 92- 7/ 3 18:01
TIMEA.ASM ---------------      2823 92- 7/14 18:08
TIMES.ASM ---------------      3662 92- 7/ 3 18:01
Free memory   185504.
```

ファイルのバイナリー表示（ダンプ）

ファイルのバイナリー表示（漢字）

手順は『ファイルのテキスト表示』と同じです．

ファイルの内容を16進数で表示します．

```
ZZ386 ミニ版 for 8086, Ver 1.0 (c)Micro-org 1992 [CA.ZPJ]          8086
                ファイル [ FS.EXE ] の内容表示
00000:4D 5A D3 00 04 00 01 00 20 00 01 04 FF FF 4E 00    MZ..... .....N.
00010:00 40 00 00 00 00 00 00 3E 00 00 00 01 00 FB 50    .@......>......P
00020:6A 72 00 00 00 00 00 00 00 00 00 00 00 00 00 00    jr..............
00030:00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00    ................
00040:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00050:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00060:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00070:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00080:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00090:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
000A0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
000B0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
000C0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
000D0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
000E0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
000F0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00100:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00110:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00120:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00130:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00140:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00150:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
↓↑ カーソル移動:   RollUp 次画面:   RollDown 前画面:        [ ESC ] キャンセル
```

ターゲットのダンプ

機能は「ファイルのバイナリー表示」と同じですが，現在解析中のプログラムのロード・モジュールのみを16進表示します．

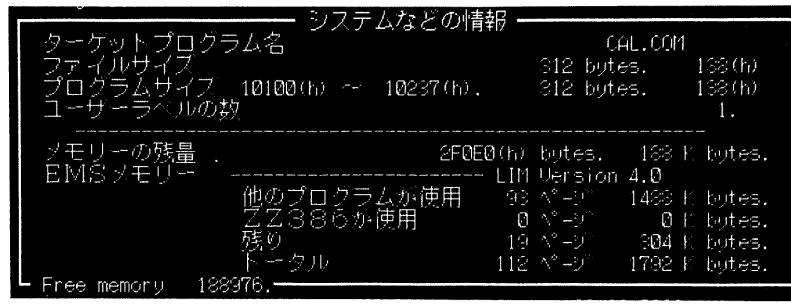
プリンタ出力の設定（行数，幅）

逆アセンブルで得られたソース・プログラム，クロス・リファレンスのデータはプリンタに出力することができます．デフォルトはつぎの設定になっています．

```
プリンターの行 幅
L: 1ページの行数 :  60
C: 1行の桁数     : 136
Free memory   192240.
```

システムの情報

ターゲットのサイズ，メモリ，EMSの情報などが表示されます．

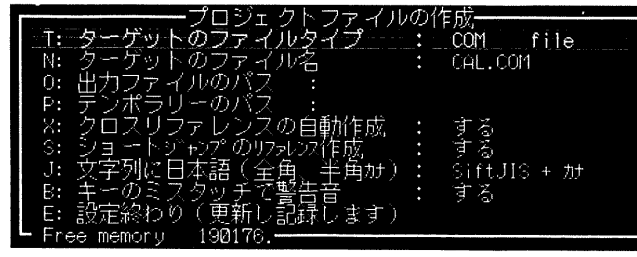


プロジェクトファイルの表示，変更

プロジェクト・ファイルの設定変更

プロジェクト・ファイルの諸設定

を参照してください．



解析開始点の追加，変更，削除

ZZ386はプログラムを疑似的にトレースし，プログラムを解析しています．通常はすべてのエリアをもれなく解析するのですが，次の理由などで未解析エリアが生じることがあります．

①割り込みベクタのセット方法が複雑で，割り込みプログラムのアドレスが検知できなかった．

②メモリ間接ジャンプで，ジャンプ・アドレスのセット方法が複雑でジャンプ先アドレスが検知できなかった．

このように未解析エリアが生じたときに，第2，第3の解析開始点を指定して未解析エリアをなくすことができます．

＊付属のディスケット内のファイルではこの処理は必要ありません．

《解析開始点》のアドレス

最初に「解析開始点のアドレス」をセグメント：オフセットの形式で入力します．

《FAR，NEAR》

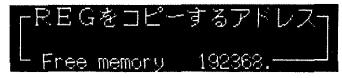
プロシージャの距離を指定します．

《解析時のレジスタなどの初期化》

メモリ間接ジャンプでは，ジャンプしている行のレジスタ値に初期化して解析をする必要があります．

割り込みプログラムではCS以外は無視した方が良い結果が得られます．

「現在位置の値を使用する 内部エントリー」を選択すると，レジスタなどをコピーするアドレスを入力します．

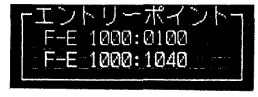


以上で解析開始アドレスの追加ができました．

画面の右上には追加された解析開始アドレスが表示されます．

最下行のガイドには「追加」，「変更」，「削除」のガイドが表示されます．

追加された解析開始アドレスの表示例



EXEヘッダの情報表示

ターゲットがEXE，EXPタイプのときはEXEヘッダの情報を表示することができます．

EXEヘッダは識別子である２文字（MZ）から始まり，ファイルのサイズ，リロケーション・データ，セグメント初期値などの情報が含まれています．

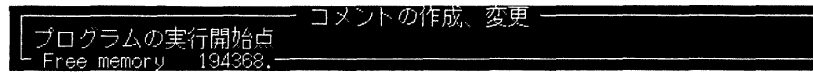
コメント

f-3 または C でコメントのメニューが表示されます.

コメントのセット：

カーソル行に新しくコメントを作成します.

カーソル行にすでにコメントがあるときは，それを変更します.
コメント文は表示され，１文字の修正なども可能です.



コメントの削除：

カーソル行のコメントを削除します．カーソルがコメント行にないときこのコマンドは無視されます.

ラベル

ラベルのセット，ラベルの削除，解釈の変更をおこないます.
カーソルがオペコード側にあるときはラベルの定義，カーソルがオペランド側にあるときは参照について変更をおこないます.

f-4 または L で実行します.

カーソルがオペコード側

カーソルがある命令行にラベルをセット，削除をおこないます．カーソルが命令行にないとき（コメント行とか）はこのコマンドは無視されます.

メニューから選択します.

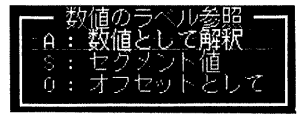
ユーザー・ラベル
User Label

ユーザーラベルをセットを選択したときはラベル名の入力をします.

カーソルがオペランド側

数値の解釈を変更します．命令行が数値を含まないときは無視されます．

メニューから処理を選択します．



［オフセットの例］

次の部分は文字列をCRTに出力しています．**DX**に文字列のアドレスをセットして**AH=09H**で**int21h**を発行しています．

DXにセットされている**03AFh**は数値ではなく文字列のアドレス（オフセット）として解釈すべきですね．

```
ZZ386 ミニ版 for 8086. Ver 1.0 (c)Micro-org 1992 [GOMI.ZPJ]        8086
C1000:0124          JNB     CI_10132                    ;73 +0C
C1000:0126          INC     AL                          ;FE C0
C1000:0128          MOV     DL,AL                       ;8A D0
C1000:012ACI_1012A: CALL    Near Ptr CI_1013B           ;E8 000E
C1000:012DCI_1012D: MOV     AX,04C00H                   ;B8 4C00
          子プロセスの終了 AH=4C AL=Ret Code.
C1000:0130          INT     021H                        ;CD 21
C1000:0132CI_10132: MOV     DX,Offset ES_103AF          ;BA 03AF
C1000:0135          MOV     AH,009H                     ;B4 09
          文字列の出力 AH=09 文字列(DS:DX)-->
C1000:0137          INT     021H                        ;CD 21
C1000:0139          JMP     Short CI_1012D              ;EB +F2
          DSIZE  endp
          Prog_3  proc  Near
C1000:013BCI_1013B: MOV     DS:[EB_1044A],DL            ;88 16 044A
C1000:013F          MOV     AH,036H                     ;B4 36
          ディスクの空き容量 AH=36 DL(0=Cur,1=A:,2=B:,...
C1000:0141          INT     021H                        ;CD 21
C1000:0143          CMP     AX,0FFFFH                   ;3D FFFF
C1000:0146          JZ      CI_1015C                    ;74 +14
C1000:0148          MOV     DS:[EW_1044B],AX            ;A3 044B
C1000:014B          MOV     DS:[EW_1044D],BX            ;89 1E 044D
C1000:014F          MOV     DS:[EW_1044F],CX            ;89 0E 044F
 End   Option Coment Label  Jump          Search segmnt X-fer  Tnasm  Make
```

カーソルを**DX**の行のオペランド側に合わせ，L を押します．

メニューから「**オフセット**」を選択する．

現在評価されているセグメント値（この場合**DS**）が表示されます．

これで良ければ **RETURN** を押します．変更の必要があれば変更します．

に表示は変わります.

注：被参照側**1000:03AF**にすでにラベルがある必要があります.
　ラベルがないときは参照属性は変更されますが，表示は変わりません.

「セグメント値」はセグメント値への解釈変更に使用します.

MOV AX, SEG SEG_NAME

などのプログラムに使います.

ジャンプ

　プログラムを解析するときには，関連するところや特定の場所を眺めたいことがよくあります.
　ZZ386では豊富なジャンプ・コマンドとバック・ジャンプによりプログラムを見失うことなくプログラムの追跡が可能です.

f-5 または J を押すとメニューが表示されます.
メニュー内にはジャンプ先のアドレスが共に表示されます.

絶対値へのジャンプ

ジャンプ先のアドレスを直接指定します.

形式は実アドレスまたは「セグメント：オフセット」です.

10200
1000:200

の形式です.

指定されたアドレスが命令コードの中間を指しているときは，そのアドレスを含む命令から表示されます.

オペランドへのジャンプ

カーソルが指す命令行のオペランドが2バイトの数値を含むときは，その実行的なアドレスへジャンプします.

命令行がジャンプ，コール命令であればジャンプ先のプログラムへジャンプします.

各セグメント＋オフセット

カーソルが指す命令行のオペランドが2バイトの数値を含むとき，カーソル行での各セグメントレジスタの評価値＋2バイトの数値でジャンプします.

それぞれ,

CS+2バイトの数値
DS+2バイトの数値
SS+2バイトの数値
ES+2バイトの数値

マルチ・セグメントのプログラム（ラージ・モデルのEXEファイルなど）で非常に有効です.

マルチ・セグメント
multi segment

数値をセグメントベース見立て

カーソルが指す命令行のオペランドが2バイトの数値を含むとき，その数値がセグメント・グループのベース・セグメントと一致すれば，そのセグメント・グループの先頭にジャンプします.

ベース・セグメント
base segment

マルチ・セグメントのプログラム（ラージ・モデルのEXEファイルなど）で非常に有効です.

前の位置へジャンプ（バック・ジャンプ）

ジャンプ機能の要になる機能です.

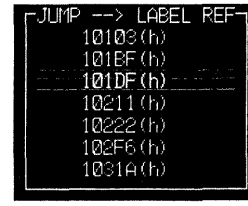
ジャンプを実行するとジャンプ元のアドレスが順次記憶されます.

バック・ジャンプはその記憶を逆にたどって戻っていきます.

ラベル参照位置へのジャンプ

カーソルがオペコード側にあるときは，そこで宣言してあるラベルを参照参照しているアドレスが一覧表示されます．

カーソルがオペランド側にあるときは，そこで参照しているのと同じラベルを参照しているアドレスが一覧表示されます．

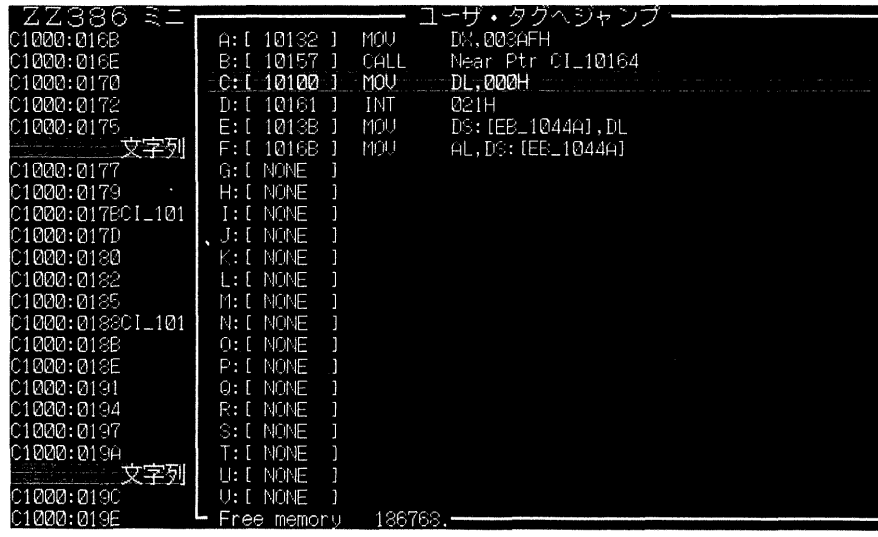


ユーザー・タグ
User Tag

ユーザータグへジャンプ

セットされているユーザー・タグが一覧表示されます．

アドレスとその行のコードが表示されます．

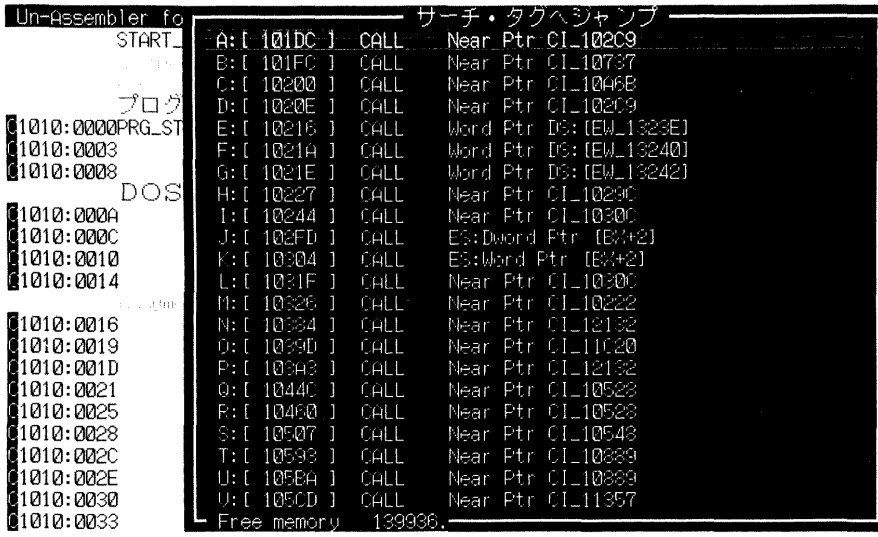


参照　カーソル行をタグにセット，削除

サーチタグへジャンプ

サーチ結果はサーチ・タグに記憶できます．サーチ・コマンドと共に使用します．

アドレスがコードと共に一覧表示されます．

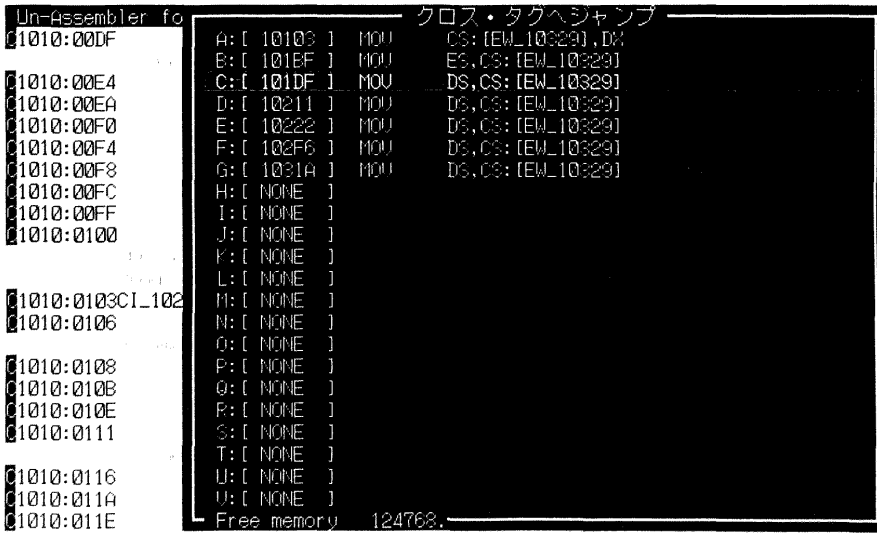


参照　サーチ

クロスタグへジャンプ

クロス・タグ
Cross Tag

クロス・リファレンスを表示し，表示を閉じたときのカーソル行のデータが自動的にクロス・タグに記録されます．



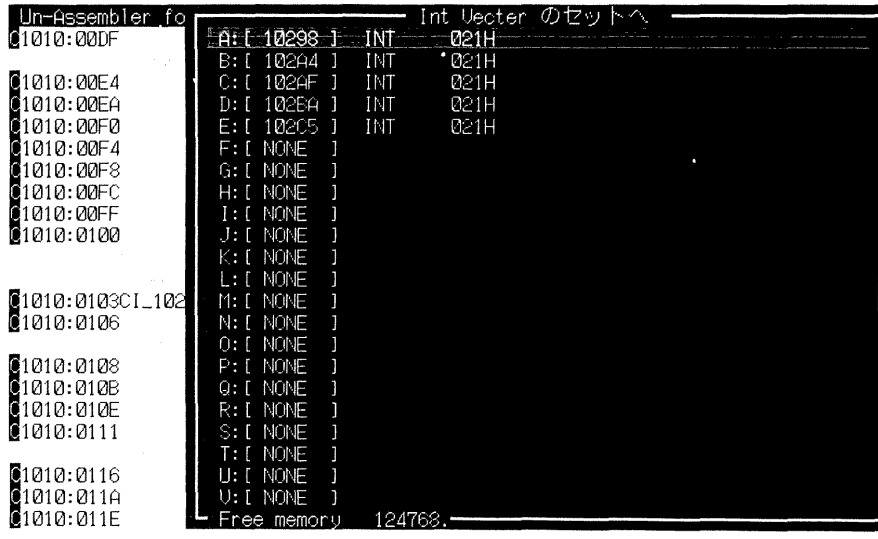
参照：機能的には「ラベル参照位置へのジャンプ」と似ています．

割り込みベクタをセットしている部分

プログラム解析の基準となるもののひとつに，割り込みベクタを変更している部分を見つけるここがあります．

メイク（自動解析）を実行するとプログラム中のこれらの個所が自動的にマークされます．

これらの個所へのジャンプです．



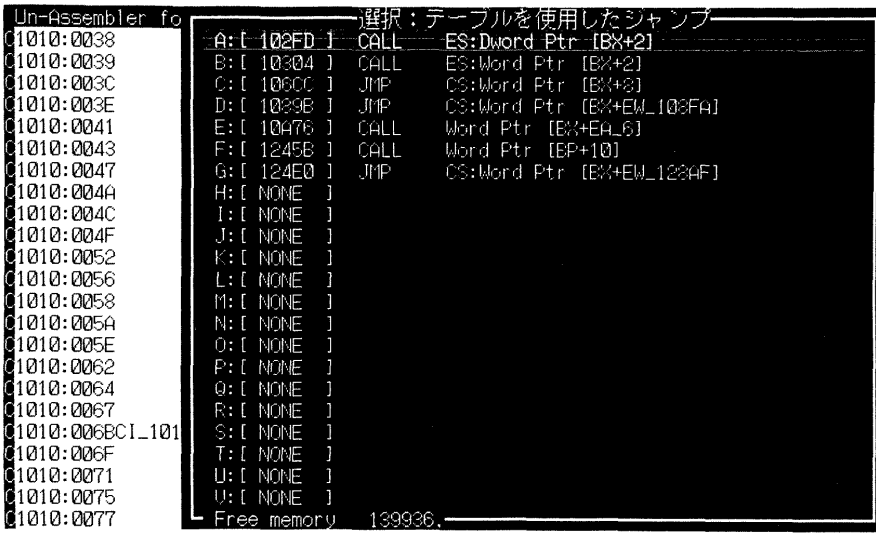
ジャンプ・テーブルを使用している部分

ジャンプ・テーブル
Jump Table

プログラム解析において困難さの原因の1つであり，また多くの情報を得られるものにジャンプ・テーブルを使用したCALL，JMPがあります．

メイク（自動解析）を実行すると，プログラム中のこれらの個所が自動的にマークされます．

プログラムによってはこれらのデータが非常に多くあり，1画面に表示しきれないことがあります．そのためカーソル行以降（アドレスが大きくなる方向）のデータのみが1画面分表示されます．



CS:IPの初期値

プログラムの実行開始点へジャンプします．

COMファイルでは「**1000:0100**」へ，EXE，EXPファイルでは，EXEヘッダに記録されている「**CS:IP**」の初期値にジャンプします．

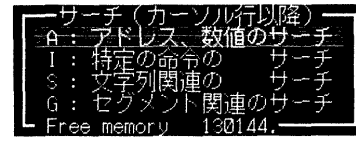
デバイス・ドライバではプログラムの先頭へジャンプします．

サーチ

サーチ機能は多くのサーチ・モードを持っています.

サーチ結果はサーチ・タグに記録でき，その結果をジャンプ機能で利用することにより，効果的でスピーディな解析ができます.

f·6 または S を押すとジャンルを決めるメニューが表示されます.



アドレス，数値のサーチ

特定のアドレス，数値を扱っている命令行を検索します.

メニューより選択します.

検索基準数値が**2000H**とすると，

□ジャンプ，コール

```
JMP 2000H
CALL 2000H
```

などの命令行を検索します.

ロード・バリュー
Load Value

□ロードバリュー

```
MOV AX, 2000H
MOV SI, 2000H
```

などの命令行を検索します.

□ダイレクトアドレス

ダイレクト・アドレス
Direct Adress

MOV AX,DS:[2000H]

などの行を検索します.

特定の命令のサーチ

特定の命令グループによる検索です.

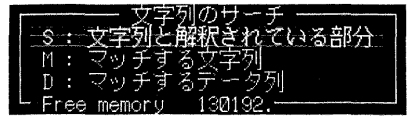
メニューから選択します.

文字列関連のサーチ

文字, データ列の検索です.

マッチする文字列では検索する文字列をテキスト（文字）で指定します.

マッチするデータ列では検索するデータ列を16進数で指定します.



セグメント関連のサーチ

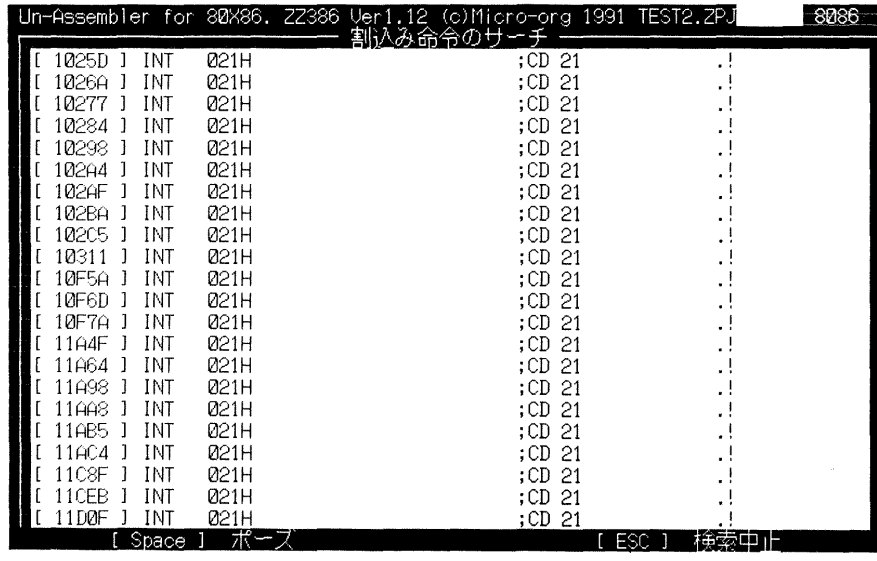
マルチ・セグメントのプログラムを解析するとき，セグメント値を変更している場所は重要なポイントになります．

すべてのセグメント・レジスタまたは特定のセグメント・レジスタに着目して検索します．メニューで選択します．

サーチ・タグのセット

SPACE キーで検索をポーズ（1時停止）します．
ESC キーで検索をキャンセル（中止）します．

検索が終了したときまたはポーズ中に T を押すと現在表示されている検索結果がサーチ・タグに記録されます．



セグメント値

86系のプログラムを逆アセンブルするとき，特にZZ386ではプログラムをトレースしているので，セグメント値が重要な意味を持ちます．

マルチ・セグメントのプログラムでは特に重要になります．多くの場合自動解析だけで充分で，セグメント値の変更は必要ありませんが，まれに自動解析中に誤解することがあります．

f·7 または G で各セグメント・レジスタのカーソル行での評価値が表示されます．

カーソルを変更するセグメントに合わせ RETURN を押すと変更モードになります．

関連：セグメント評価値の表示，非表示

クロス・リファレンス

クロス・リファレンスのコマンドはCRT，ファイル，プリンタへの出力とクロス・リファレンス・データの作成です．

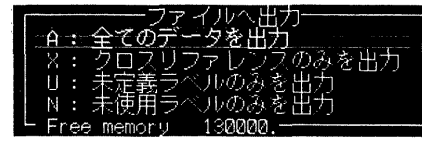
データはメイク時に自動作成するようにもプロジェクト・ファイルで設定できます．

f-8 または X でメニューが表示されます．

CRT，ファイル，プリンタのいずれかを選択すると表示（出力）するデータを選択します．

ＣＲＴの例

ファイル，プリンタの例

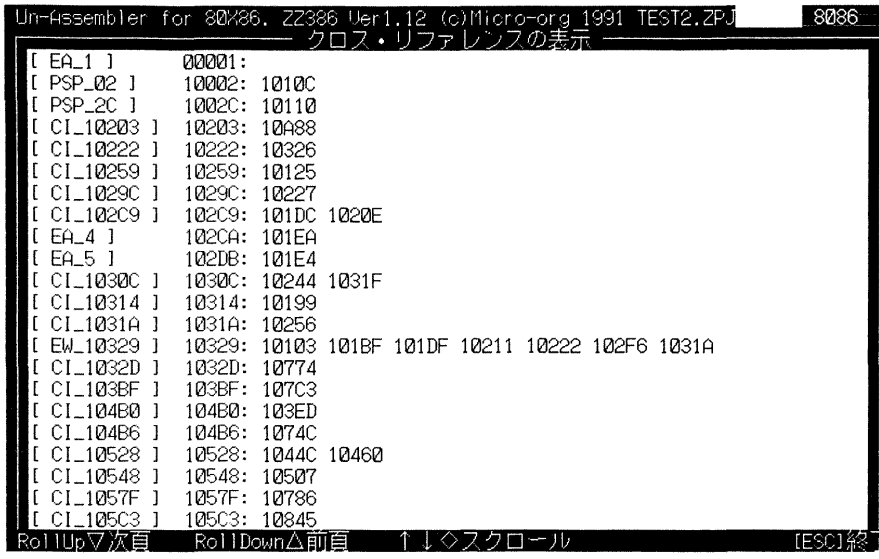


クロス・リファレンスの生成

クロス・リファレンスに関連する3つのデータが作成されます.

作成中に表示されるメッセージ

クロスリファレンスの表示例



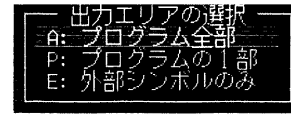
逆アセンブルの出力

逆アセンブルで作成されたソース・プログラムはファイルまたはプリンタに出力できます.

f-9 または U でメニューが表示されます.
出力先（ファイルまたはプリンタ）を選択します.

範囲指定

出力先を選選択すると，範囲指定のメニューが表示されます.



通常は「プログラム全部」を選択すればよいのですが，1部のサブ・ルーチンだけとかを出力したいときは，「プログラムの1部」を選択します.

「プログラムの1部」を選択すると，画面の最下行にガイドが表示されます.

①開始点のガイドが表示されます.
②カーソルを開始点に合わせます.
③ RETURN で開始点が決定されます.
④終了点のガイドが表示されます.
⑤カーソルを終了点に移動します.
⑥ RETURN で終了点が決定されます.
⑦ガイド表示が消えます.
⑧出力されます.
ESC で中止されます.

最下行のガイド＝開始点の決定

```
Un-Assembler for 80X86. ZZ386 Ver1.12 (c)Micro-org 1991 TEST2.ZPJ        8086
C1010:00DF          MOV     DS,CS:[EW_10329]              ;2E: 8E 1E 0229

C1010:00E4          MOV     CS:Byte Ptr [EA_5],072H       ;2E: C6 06 01DB 72
C1010:00EA          MOV     CS:Byte Ptr [EA_4],000H       ;2E: C6 06 01CA 00
C1010:00F0          PUSH    Word Ptr DS:[EW_12F96]        ;FF 36 0086
C1010:00F4          PUSH    Word Ptr DS:[EW_12F94]        ;FF 36 0084
C1010:00F8          PUSH    Word Ptr DS:[EW_12F92]        ;FF 36 0082
C1010:00FC          CALL    Near Ptr CI_10737             ;E8 0538
C1010:00FF          PUSH    AX                            ;50
C1010:0100          CALL    Near Ptr CI_10A6B             ;E8 0868


C1010:0103CI_10203: MOV     AX,Seg Seg_4                  ;B8 12F0
C1010:0106          MOV     DS,AX                         ;8E D8

C1010:0108          MOV     SI,00002H                     ;BE 0002
C1010:010B          MOV     DI,00002H                     ;BF 0002
C1010:010E          CALL    Near Ptr CI_102C9             ;E8 00B8
C1010:0111          MOV     DS,CS:[EW_10329]              ;2E: 8E 1E 0229

C1010:0116          CALL    Word Ptr DS:[EW_1323E]        ;FF 16 032E
C1010:011A          CALL    Word Ptr DS:[EW_13240]        ;FF 16 0330
C1010:011E          CALL    Word Ptr DS:[EW_13242]        ;FF 16 0332
 [RETURN] 開始点指定:  逆アセンブル出力の範囲指定  ↑↓◇カーソル移動  [ESC]中止
```

最下行のガイド＝終了点の決定

```
Un-Assembler for 80X86. ZZ386 Ver1.12 (c)Micro-org 1991 TEST2.ZPJ        8086
C1010:00DF          MOV     DS,CS:[EW_10329]              ;2E: 8E 1E 0229

C1010:00E4          MOV     CS:Byte Ptr [EA_5],072H       ;2E: C6 06 01DB 72
C1010:00EA          MOV     CS:Byte Ptr [EA_4],000H       ;2E: C6 06 01CA 00
C1010:00F0          PUSH    Word Ptr DS:[EW_12F96]        ;FF 36 0086
C1010:00F4          PUSH    Word Ptr DS:[EW_12F94]        ;FF 36 0084
C1010:00F8          PUSH    Word Ptr DS:[EW_12F92]        ;FF 36 0082
C1010:00FC          CALL    Near Ptr CI_10737             ;E8 0538
C1010:00FF          PUSH    AX                            ;50
C1010:0100          CALL    Near Ptr CI_10A6B             ;E8 0868


C1010:0103CI_10203: MOV     AX,Seg Seg_4                  ;B8 12F0
C1010:0106          MOV     DS,AX                         ;8E D8

C1010:0108          MOV     SI,00002H                     ;BE 0002
C1010:010B          MOV     DI,00002H                     ;BF 0002
C1010:010E          CALL    Near Ptr CI_102C9             ;E8 00B8
C1010:0111          MOV     DS,CS:[EW_10329]              ;2E: 8E 1E 0229

C1010:0116          CALL    Word Ptr DS:[EW_1323E]        ;FF 16 032E
C1010:011A          CALL    Word Ptr DS:[EW_13240]        ;FF 16 0330
C1010:011E          CALL    Word Ptr DS:[EW_13242]        ;FF 16 0332
 [RETURN] 終了点指定:  逆アセンブル出力の範囲指定  ↑↓◇カーソル移動  [ESC]中止
```

ファイル名の確認と変更

ソース・プログラム，インクルード・ファイル名はプロジェクト・ファイルの名前にそれぞれの拡張子が付けられます．

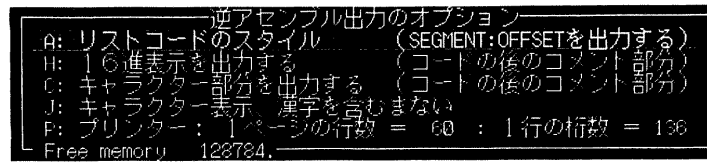
確認のため表示されるので，必要ならばこのとき変更します．

```
┌ ファイル名の確認と変更 ┐
CA.ASM
└ Free memory   130816.┘
```

```
┌ ファイル名の確認と変更 ┐
CA.EQU
└ Free memory   130816.┘
```

オプション（リストの形式）

「オプション」を選択するとメニューが表示されます．



出力する形式は用途により，ソース・コードのスタイル（アセンブル可能），リスト・ファイルのスタイル（アドレス情報を含みアセンブル不可）を選択できます．

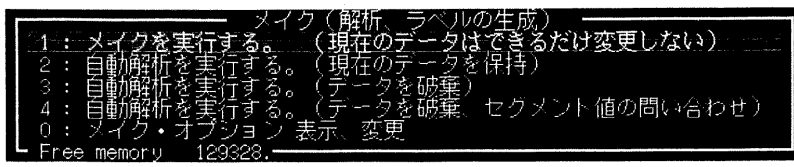
またファイル・サイズを小さくするため一部の情報を出力しないように設定できます．

```
MOV  AL,ES:Byte Ptr [EA_1]          ; 26: A0 0000 &...
                                    ~~~~~~~~~~~~ ~~~~~
この部分が「１６進表示を出力する」    ↑          ↑
                                    この部分が「キャラクタ部分」
```

自動解析

f·10 または M でメニューが表示されます.

標準の自動解析はモード3です. 3 を押してください.



①は，ほとんど解析をおこないません.

②は，ユーザの設定したセグメント値などのデータを使用し，データ解析はおこなわれません.

④は，モード③＋セグメント値の確認がおこなわれます.

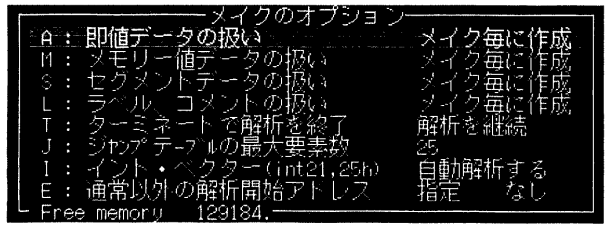
実際は自動解析の過程を確認するのが面白い程度かと思います.

モード③とマニュアル操作を併用するのが標準的で効率の良い使用法だと思います.

オプション

メイクで作成されたデータの扱い，処理の内容を設定します.

メニューより項目を選んで変更します.



《即値データの扱い》

メイク毎に作成，保持するのいずれかを選択します.

```
MOV     AX,078BH
```

などの数値をロードしている各命令行においてデータが作成されます.

このデータはその数値が即値，オフセット値，セグメント値のいずれであるかの情報を記録しています.

「メイク毎に作成」を選択すると，自動解析を始めるときに古いデータはすべて破棄されます.

「保持する」を選択すると，データが存在するところでは古いデータが使用されます.

《メモリ値データの扱い》

「メイク毎に作成」，「保持する」のいずれかを選択します．

```
MOV       AX, DS:[078BH]
```

などのメモリから数値をロードしている各命令行においてデータが作成されます．

《セグメントデータの扱い》

「メイク毎に作成」，「保持する」のいずれかを選択します．

マルチ・セグメントのデータ，セグメント評価値のデータです．

《ラベル，コメントの扱い》

「メイク毎に作成」，「保持する」のいずれかを選択します．

「メイク毎に作成」を選択すると，ユーザラベルを除くデータが破棄されます．

《ターミネートで解析を終了》

「解析を継続」，「終了する」のいずれかを選択します．

```
AH=00 , 31h , 4Ch
int     21h
```

```
int     20h
```

```
int     27h
```

などのプログラムを終了するコードを解析したときの処理です．

自動解析において，これらを解析したときは1つの解析の流れを終了するべきなのですが，コンパイラで作成された実行ファイルでは「終了ルーチン」をコールしていることが多く，「解析を継続」にした方が良い結果が得られます．

《ジャンプテーブルの最大要素数》

```
                JMP       CS:[BX+1000h]
                          ～～～～
1000:1000       DW        1100h
                DW        1200h
                          ～～～～
```

などのテーブルを使用したジャンプ，コールの解析で，解析を試みる最大テーブル・データ数を指定します．

デフォルトは25です．

《割り込みベクタ(int21,25h)》

「自動解析する」，「レポートする」のいずれかを選択します．

自動解析のとき，割込ベクタのセット（AH=25h,int21h）を解析すると，ZZ386はこの部分を記録します．

この記録はジャンプ機能の「イントベクターをセットしている部分へのジャンプ」で使えます．

「レポートする」を選択すると，これ以上の処理はおこなわれません．

「自動解析する」を選択すると，割り込みプログラムのアドレスは解析開始点に自動的に追加され，その割り込みプログラムも解析されます．

《通常以外の解析開始アドレス》

オプション・メニューからの「解析開始点の追加，変更，削除」と同じです．

簡単な変更コマンド（主にマニュアル操作）

それ以外に頻繁に使用するコマンドはアルファベット・キーなど使いやすいキーに割り当てられています．

カーソル行の属性変更

これらの命令は自動解析によらず，いわばマニュアル操作で解析を進めるのに使います．ためしに解釈の変更をおこなったりできますので，難解な部分などをコード⇔データと変更し意味を確定することが簡単にできます．

また自動解析の誤りを画面をみながら訂正でき，アドレス（オフセット）の計算などもZZ386がやってくれます．出力されたソース・プログラムをエディタで修正するのと比較して，効率良くソース・プログラムの完成度を上げることができます．

命令行に解釈

カーソルのある行を命令として解釈します．

アルファベットの[I]を押します．

カーソルは次の行に移動します．

```
C1000:0115          JZ      CI_10148            ;74 +31
```

バイト・データに解釈

カーソルのある行の先頭をバイト・データに解釈します．

[1]，[B]のどちらかを押します．

カーソルは次の行に移動します．

```
C1000:0115          DB      074H                ;t
```

ワード・データに解釈

カーソルのある行の先頭をワード・データに解釈します．

[2]，[W]のどちらかを押します．

カーソルは次の行に移動します．

```
C1000:0115          DW      03174H              ;3174
```

ダブルワード・データに解釈

カーソルのある行をダブルワード・データに解釈します．

[4]，[D]のどちらかを押します．

カーソルは次の行に移動します．

```
C1000:0115          DD      0C0333174H          ;C0333174
```

クォードワード・データに解釈

カーソルのある行をクォードワード（8バイト）・データに解釈します．

[8]，[Q]のどちらかを押します．

カーソルは次の行に移動します．

```
C1000:0115          DQ      0743133C08EC026A1H  ;
```

Tワード・データに解釈

カーソルのある行をTワード（8バイト）・データに解釈します．

[0]，[T]のどちらかを押します．

カーソルは次の行に移動します．

```
C1000:0115          DT      0743133C08EC026A1BA00H          ;
```

ブロック単位の属性変更 … RETURN

「カーソル行の属性変更」のブロック単位版と考えればよいのですが，DUP，文字列のようにブロック単位でしか変更できない属性があります．

カーソルをブロックの先頭に合わせ **RETURN** を押すと，「ブロック・モード」になり属性を選択するメニューが表示されます．

どれかの属性を選択し，カーソルをブロックの最後に移動します．

再び **RETURN** を押すと，その範囲の属性が変更されます．

変更の手順

①ブロックの開始	**RETURN**
②属性の選択	メニュー
③ブロックの最後	カーソル移動
④決定	**RETURN**

途中でやめたいときは **ESC** を押します．

キャラクタ部分の表示…HOME CLR

画面の右側にはプログラムが16進数で表示されています．

慣れてくるとこれを見るだけで「文字列ではないか？」とか分かるのですが，色々な表示形態に変えられた方が解析のヒントも得やすいものです．

標準の表示

```
                        ↓＋は符合付きデータを表す
C1000:0115   JZ      CI_10148                ;74 +31
C1000:0117   XOR     AX,AX                   ;33 C0
C1000:0119   MOV     ES,AX                   ;8E C0
          2バイトデータはハイ，ローが入れ替えられて表示
C1000:011B   MOV     AX,ES:Word Ptr [EA_5]   ;26: A1 00BA
                     ↑：はプリフィックスを表す
```

簡潔な順序通りの表示

```
C1000:0115   JZ      CI_10148                ;7431
C1000:0117   XOR     AX,AX                   ;33C0
C1000:0119   MOV     ES,AX                   ;8EC0
C1000:011B   MOV     AX,ES:Word Ptr [EA_5]   ;26A1BA00
                   アドレス順に加工されずに表示↑
```

文字として表示（英数，半角）

```
C1000:0115   JZ      CI_10148                ;t1
C1000:0117   XOR     AX,AX                   ;3ﾀ
C1000:0119   MOV     ES,AX                   ;.ﾀ
C1000:011B   MOV     AX,ES:Word Ptr [EA_5]   ;&｡ｺ.
```

表示可能な文字は文字として，文字にならない数値は‘.’で表示されます．

プロジェクト・ファイルで「英数のみ」を指定したときは，半角カナの部分も全角文字も文字として表示されます．

2行にまたがって全角文字と解釈されるときは前の行に表示されます．

文字として表示（英数，半角，全角）

```
C1000:0115  JZ      CI_10148                ;tl
C1000:0117  XOR     AX,AX                   ;3ﾀ
C1000:0119  MOV     ES,AX                   ;実
C1000:011B  MOV     AX,ES:Word Ptr [EA_5]   ;&｡ｺ.
C1000:011F  MOV     DS:[EW_1089C],AX        ;｣..
```

HOME CLR キーを押すごとに

標準 → 短縮 → 英数・半角 → 英数・全角

と順次表示が切り替わります．

セグメント評価値の表示非表示 … [ ]

マルチ・セグメントのプログラム（EXEファイルなど）の解析ではセグメント値が正しく解釈されているかどうかが重要となります．

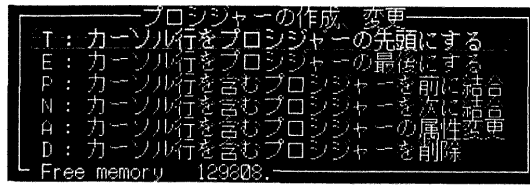
[ ]または[ ]を押すとカーソル行でのセグメントの解釈値がいつも表示されます．もう1度押すと表示が消えます．
ニーモニックの表示のジャマにならないよう画面の右端に表示されます．

＊セグメント解釈値の変更はセグメント・コマンドを使用します．

プロシージャの変更 … P

自動解析で生成されたプロシージャを分割したり，結合したりします．

P を押します．
メニューから処理を選択します．
作成，属性変更ではFAR，NEARを選択するサブメニューが表示されます．



ロード値，ジャンプテーブルの表示，変更 … V

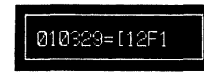
ZZ386ではプログラムの自動解析ををおこなうのにプログラムを疑似的にトレースしています．このときレジスタの値やメモリーの値が重要な意味を持つことがまれにあります．

```
C1000:011B    MOV    AX,ES:Word Ptr [EA_5]        ;26: A1 00BA

C1000:011F    MOV    DS:[EW_1089C],AX             ;A3 089C
```

などの行では前者はメモリからの値，後者はAXレジスタの値を保持しています．
V を押すと現在の評価値を表示します．

```
                                          ↓表示
C1000:011B    MOV    AX,ES:Word Ptr [EA_5]        ;26: A1 00
C1000:011F    MOV    DS:[EW_1089C],AX             ;A3 089C AX=[0000
C1000:0122    MOV    AH,052H                              ;B4 52
```

このとき必要であれば数値を変更します．
ESC を押すと変更されません．
カーソルがジャンプテーブルを使ったJMP，CALL命令ををを指しているときは現在のテーブル解析のパラメータを表示します．



それぞれの意味は，

セグメント（テーブル）	：	テーブルのアドレス
セグメント（コード）	：	ジャンプ先コードのアドレス
オフセット	：	テーブルのアドレス
ジャンプのタイプ	：	FAR，NEAR
オフセットのサイズ	：	前記の例では2バイト
1データのサイズ	：	まれにテーブル中に呼ばれるプログラムへのパラメータを含むことがあります．
テーブルのデータ数	：	最大解析テーブル要素数

カーソル行をタグにセット，削除… T

プログラムを解析中は参照したい場所が多く出てくるものです．それらを覚えておくのは大変ですし，面倒なものです．それらの箇所にマークを付けておき，瞬時にその場所にジャンプする機能です．

T を押すと「設定または削除」を選択するメニューが表示されます．
どちらかを選択すると「現在設定されているタグ」の一覧が表示されます．

CBW CWD ESC
IN JCXZ
HF LDS LEA
CK LODS LOOP
OP NOT OUT
SH PUSHF REP
G STOS WAIT

第5部　実行ファイルの種類

実行ファイルには「COMファイル」,「EXEファイル」,「デバイス・ドライバ」があります．デバイス・ドライバは直接コマンド・ラインからは実行できませんが，一応仲間に入れておきましょう．COM, EXEの変形として常駐型プログラムがあります．

XCHG XLAT
DD CMP CMPS
BB SCAS SUB
OR SAR SHL
TEST XOR
INC
CR ROL ROR

1　COMファイル，EXEファイル

これらについてはいろいろと述べてきたので，ここでは「EXEヘッダ」の形式について説明します．

ヘッダ
Header

『ZZ386』では，オプション・コマンド内の「EXEヘッダの情報表示」で表示できます．

「EXEヘッダ」は「EXEファイル」の先頭に記録されていて，最低512バイトのサイズをもちます．

こんなことは知らなくても，プログラムを作るのには何の不自由もないので，興味のない方は次へどうぞ…．

「EXEヘッダ」の先頭は図のようにデータが入っています．

ＥＸＥヘッダの先頭のデータ構造

ファイルのアドレス	内　容
0000h～0001h	"EXEファイルの識別子（4Dh,5Ah）"
0002h～0003h	最終ページのサイズ（1ページは512バイト）
0004h～0005h	ファイルのページ数
0006h～0007h	リロケートする項目数
0008h～0009h	ヘッダのパラグラフ数 （16倍した位置からロード・モジュールが始まる）
000Ah～000Bh	最小必要メモリ（パラグラフ単位）
000Ch～000Dh	最大必要メモリ（パラグラフ単位）
000Eh～000Fh	SSのオフセット
0010h～0011h	SPの初期値
0012h～0013h	SUM
0014h～0015h	IPの初期値
0016h～0017h	CSのオフセット
0018h～0019h	リロケーション・テーブルの位置
001Ah～001Bh	オーバーレイ番号

《ファイル・サイズ》

ファイル・サイズは次のように求められます．

（全ページ数-1）×1ページのサイズ	＋	最終ページのサイズ
[0004h]	512 バイト	[0002h]

《ロード・モジュール（プログラム本体）の位置》

0008hの位置にヘッダのサイズがパラグラフ単位で記録されています．

```
00000000  4D 5A 39 00 02 00 01 00-20 00 21 00 FF FF 04 00   MZ9..... .!.....
00000010  00 02 00 00 00 00 00 00-3E 00 00 00 01 00 FB 30   ........>......0
00000020  6A 72 00 00 00 00 00 00-00 00 00 00 00 00 00 00   jr..............
00000030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 01 00   ................
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000090  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000000F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
000001F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

《リロケーション・データ》

リロケーション・データはリロケーション・テーブルの位置**[0018h]**からリロケートする項目数個**[0006h]**のデータが，1つが4バイトの大きさで記録されています．

「**SAMPLE41.EXE**」の例ではリロケート数が0001，リロケート・データの位置は003Ehです．

リロケート
Relocate

そのデータは0001（オフセット），0000（セグメント）となっています．この数値はリロケートすべきセグメント値のロード・モジュールの先頭からの位置を表わしています．

データは「セグメント:オフセット」の形式なので1メガバイトまでの位置を表現できます．なぜ単純な4バイト（ダブル・ワード）の形式でないのか，疑問の残るところです．

この例では，

```
        ↓ここからロード・モジュール 20h×10h=200h
00000200 B8 01 00
```

の部分が変更されるセグメント値です．

PSPがセグメント値1000hに作成されたときと，253Ehに作成されたときの例を考えてみましょう．

ロード・モジュールはそれぞれ「1010:0000」，「254E:0000」からのメモリに読み込まれます．

PSP
Program Segment Prefix
MS-DOSがプログラムを実行する際にプログラムのすぐ前の領域に作られるデータの受け渡しなどのために使うブロック

変更されるべきセグメント値は「1010:0001」，「254E:0001」にあります．元のデータはともに0001（00 01）です．

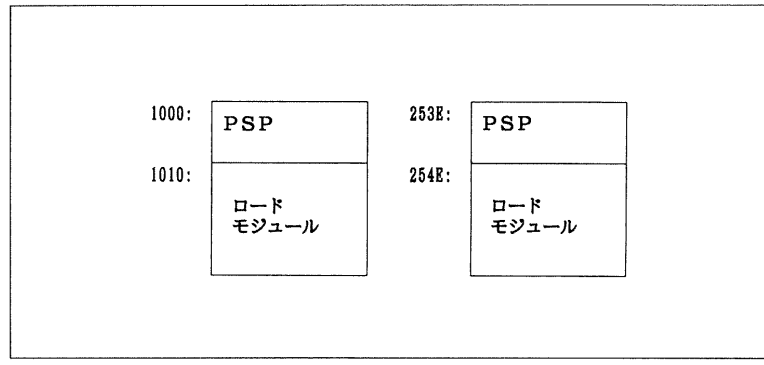
ロード・モジュールの読み込まれたセグメント値が加算されます．

それぞれ1011（11 10），254F（4F 25）の値がセットされます．

一言でいえば，EXEファイルに記録されている値はロード・モジュールがセグメント値0000からのメモリに読み込まれたとき，正常になる値がセットされているのです．

A>ZZ386 SX4.ZPJ

で確認してみてください．



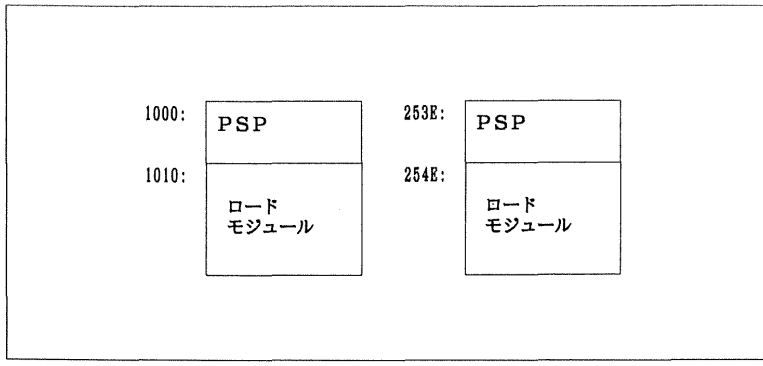

```
  0001          0001
+ 1010        + 254E
------        ------
  1011          254F
```

セグメントレジスタの初期化

EXEヘッダにはそれぞれ次の値が記録されています．

CS=0000
PC=0000
SS=0004
SP=0200

セグメント・レジスタにはリロケーション・データと同様にロードアドレスのセグメント値が加算されます．



CS：

0000	0000
＋1010	＋254E
1010	254E

SS：

0004	0004
＋1010	＋254E
1014	2553

他のセグメント・レジスタ（**ES**，**DS**）にはPSPのセグメント値がセットされます．

2 デバイス・ドライバ

デバイス・ドライバはMS-DOSに機能を付加するソフトウェアです．

ブロック型のデバイス・ドライバとキャラクタ型のデバイス・ドライバがありますが，前者はハードディスク，RAMディスク，標準では使えないフロッピーディスクの読み書きなど，ディスク関係のものと考えれば結構です．

それ以外のものは後者にあたります．身近なところでは「**PRINT.SYS**」（プリンタ），「**RSDRV.SYS**」（RS-232C）などがあります．

MS-DOSでは周辺装置を「デバイス」という概念で扱うので，「周辺装置ドライバ」と言い換えてもよいかも知れません．

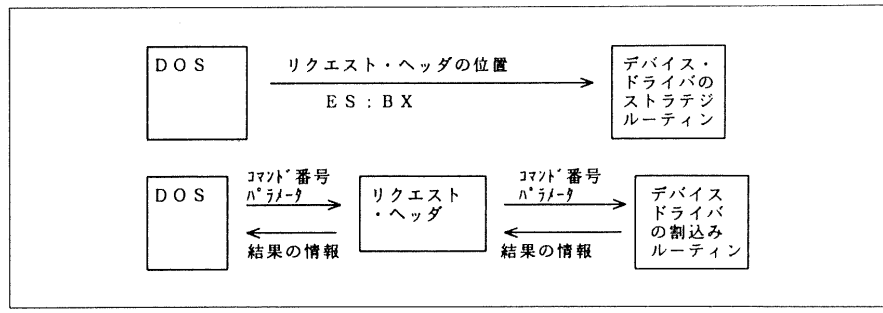
デバイス・ドライバ（ファイル）はつぎの3つの部分で構成されています．

①デバイス・ヘッダ
②ストラテジ・ルーチン
③割り込みルーチン

リクエスト
Request

最初にストラテジ・ルーチンが呼ばれ，DOSとデータの授受に使う「リクエスト・ヘッダ」というエリアのアドレスが登録され，以後のデータの受け渡しにはこの「リクエスト・ヘッダ」を使います．

デバイスドライバの動作



デバイス・ヘッダ

フォーマット
Format
書式

デバイス・ヘッダはデバイスドライバの先頭にあり，次のフォーマットになっています．

デバイスヘッダ

アドレス	内　容
00h	デバイスリンク・オフセット
02h	デバイスリンク・セグメント
04h	デバイス属性ワード
06h	ストラテジルーチン・オフセット
08h	割り込みルーチン・オフセット
0Ah～11h	名前／ユニット・フィールド

PC-9801用MS-DOS Ver5.0のPRINT.SYSのデバイス・ヘッダは次のようになっています．

```
↓オフセット
0000    DW    0FFFFH    デバイスリンク・オフセット
0002    DW    0FFFFH    デバイスリンク・セグメント
0004    DW    0A0C0H    デバイス属性ワード
0006    DW    0004FH    ストラテジルーチンのオフセット
0008    DW    0005AH    割り込みルーチンのオフセット
000A    DB    'PRN     '    論理デバイス名
```

ストラテジ・ルーチン
Strategy Routine

ストラテジ・ルーチン

ストラテジ・ルーチンのアドレスはデバイス・ヘッダのオフセット0006hに記録されています．

MS-DOSは最初にストラテジ・ルーチンを呼び出します．このときES:BXレジスタを介してリクエスト・ヘッダのポインタを渡します．

ストラテジ・ルーチンの役割はリクエスト・ヘッダのポインタを受け取り，記憶することです．

割り込みルーチン

デバイス・ドライバの本来の処理はここに記載されています.

MS-DOSはリクエスト・ヘッダにコマンド番号，必要なパラメータをセットし，割り込みルーチンを呼びます（**FAR CALL**）.

割り込みルーチンでは必要な処理をおこなった後，エラー・コードとステータスをリクエスト・ヘッダにセットしてMS-DOSに戻ります.

ステータス
Status

割り込みルーチンはリクエスト・ヘッダのコマンド番号を見て対応する処理をします.

最初にコマンド番号0（**init**）が1度だけ呼ばれます.

多くのデバイスドライバでは次のような構成となっています.

リクエストヘッダの構成

アドレス	内　容
00h	リクエスト・ヘッダの長さ
01h	ブロックデバイスのユニット数
02h	コマンド番号
03h～04h	リターンステータス・ワード
05h～0Ch	予約領域
0Dh	メディアIDバイト
0Eh～0Fh	データ転送領域のオフセット
10h～11h	データ転送領域のセグメント
12h～13h	転送バイト数/セクタ数
14h	開始セクタ番号

リクエストヘッダのステータスワード

ビット	意　味
15	エラー
12～14	予約領域
9	BUSY
8	DONE
0～7	ビット15が1のときエラー・コード

リクエストヘッダにセットされるエラーコード

番　号	意　味
00h	書き込み禁止
01h	存在しないドライブを指定
02h	ドライブが準備できていない
03h	無効なコマンドの指定
04h	CRCエラーが発生
06h	リクエスト・ヘッダの長さが不正
07h	無効なメディアを指定
08h	指定されたセクタが存在しない
09h	プリンタ用紙切れ
0Ah	書き込みエラーが発生
0Bh	読み出しエラーが発生
0Ch	何かエラーが発生
0Dh	未使用
0Eh	未使用
0Fh	不正なディスク交換

コマンド番号

コマンド番号	内　容	デバイス
0	初期化	両方
1	メディア・チェック	ブロック
2	BPB作成	ブロック
3	IOCtl入力	両方
4	読み込み	両方
5	非破壊入力（待ち時間なし）	キャラクタ
6	入力状況	キャラクタ
7	入力フラッシュ	キャラクタ
8	書き込み	両方
9	書き込み（ベリファイあり）	ブロック
10	出力状況	キャラクタ
11	出力フラッシュ	キャラクタ
12	IOCtl出力	両方
13	デバイス・オープン	両方
14	デバイス・クローズ	両方
15	交換可能メディア	ブロック
16	BUSYまで出力	両方
23	論理デバイス取得	ブロック
24	論理デバイス設定	ブロック
25	一般IOCtlリクエスト	両方

3 常駐型プログラム

常駐型プログラムは外見（動作以外）はCOMやEXE型の実行ファイルと同じです．

COMやEXE型の実行ファイルはDOSのプロンプトから実行され，終了するとDOSのプロンプトに戻ります．
終了するとプログラムの使っていたメモリはDOSに返されます．

一方，常駐型のプログラムはその名の通りメモリに残ったまま終了します．

ただメモリに残っただけでは何もできませんし，ジャマになるだけです．それではなぜメモリに残るプログラムがあるのでしょうか．

例として画面に時刻を表示するプログラムを考えてみましょう．

これを普通のプログラムで実現すると次のような構成となります．

①キー入力を監視して，E が押されると終了する
②時刻のデータを取得して表示する．

しかし，このプログラムを実行しているときは他のプログラムを使えずにパソコンがただのデジタル時計になってしまいます．

同じ事を常駐型のプログラムで実現することを考えてみましょう．

①キー入力待ちのときに「時計表示」にジャンプするような細工をする．
②時刻のデータを取得して表示する．
③プログラムを常駐する．
④プログラムの常駐を解除する．

おおよそ4つの部分から構成されます．こうすることで「時刻を表示させながら他のプログラムが使える」事になります．

このうち，

時刻のデータを取得して表示する．

以外の部分は常駐する必要はないだろうと想像できます．
では，実際に簡単なサンプルを2つ較べてみましょう．

通常のプログラム“TIMEA.COM”

“**TIMEA.ASM**”がソース・プログラムです.

COM.BATでアセンブル，リンクできます.

```
A>COM TIMEA
```

プログラムは**PROC START**から実行されます.

PROC STARTではキー入力を監視し，Ｅまたはｅが入力されるとプログラムを終了します.

それ以外では**DisplayTime**のコールを繰り返しています.

DisplayTimeでは画面がチラつくの防ぐために**DspCount**を数え，500h回に1回表示するようになっています.

表示部分は,

①エスケープシーケンスを使ってカーソルを表示個所に移動

②**AH=2Ch　int 21h**でシステム時間を取得

③文字列に変換 (**SetBcd**)

④文字列を表示

から構成されています.

```
_TEXT   SEGMENT BYTE PUBLIC 'CODE'      ;セグメントの定義
        ASSUME  CS:_TEXT, DS:_TEXT      ;CS=DS=ES=SS

        ORG     100h                    ;オフセット100hから始まる

START   PROC    NEAR
GetKeyLp:
        call    DirectKeyIn             ;キー入力の監視
        cmp     al, 'E'                 ;押されたキーが E または e なら終了
        je      EndStart
        cmp     al, 'e'
        je      EndStart
        call    DisplayTime             ;時刻を表示する
        jmp     short   GetKeyLp        ;繰り返し
EndStart:
        mov     ax, 4c00h               ;プログラムの終了
        int     21h
START   ENDP

;-----------------------------------------------------
;       キー入力ルーチン( DIRECT CON I/O )
;
;       キーが押されたらキーコードを返す
;       キーが押されないと 00  を返す
;-----------------------------------------------------
DirectKeyIn     PROC    NEAR
        mov     ah, 06h         ; DIRECT CON I/O
        mov     dl, 0ffh        ; 入力モード
        int     21h
        jnz     GetKey          ; ゼロフラグがクリアーされるとキー入力有り
        mov     al, 0           ; キー入力が無いのでクリアー
GetKey:
        ret
DirectKeyIn     ENDP
```

```
;-------------------------------------------------------
;       時刻表示ルーチン
;
;       システム時間を得て、表示する
;-------------------------------------------------------
DisplayTime     PROC    NEAR
        mov     ax, ds:[DspCount]
        dec     ax
        mov     ds:[DspCount], ax
        jz      DisplayTimeMain
        ret
DisplayTimeMain:
        ;  カーソルを移動する
        ;-------------------------------------------
        mov     bp, offset ScreenPosi
        call    OutStrZ
        ;       時刻を取得
        ;-------------------------------------------
        mov     ah, 2ch
        int     21h             ; ch=time cl=min dh=sec
        mov     bp, offset DspData
        mov     al, ch
        call    near ptr SetBcd ;時を文字列に変換
        inc     bp              ; :
        ;
        mov     al, cl
        call    near ptr SetBcd ;分を文字列に変換
        inc     bp              ; :

        mov     al, dh
        call    near ptr SetBcd ;秒を文字列に変換

        mov     bp, offset DspData      ;時、分、秒を表示する
        call    OutStrZ

        mov     ax, 500h                ; ５００ｈに１回表示する
        mov     ds:[DspCount], ax
        ret

;       画面位置設定のエスケープシーケンス
ScreenPosi      db      1bh,'[','2','4',';','7','0','H',0

;       時刻データバッファ、仮のデータが入っている
DspData         db      '12:15:30',0

;       表示のカウント
DspCount        dw      1

DisplayTime     ENDP

;-------------------------------------------------------
;       AL -> (BCD-ASCII) -> [BP]
;       BP: BCD値をセットするアドレス
;       AL: 16進データ
;-------------------------------------------------------
SetBcd  PROC    NEAR
        mov     ah, 0
SetBcdLp:
        cmp     al, 10
        jb      short   SetBcdDat
        sub     al, 10
        inc     ah
        jmp     short   SetBcdLp
SetBcdDat:
        add     ah, '0'
        mov     ds:[bp], ah
        inc     bp
        add     al, '0'
        mov     ds:[bp], al
        inc     bp
        ret
SetBcd  ENDP
```

```
;-------------------------------------------------------------
;       0で終わる文字列（ASCIIZ）　を表示する
;
;       BP = 文字列の先頭
;-------------------------------------------------------------
OutStrZ PROC    NEAR
StrLp:
        mov     dx, bp
        mov     dl, ds:[bp]
        cmp     dl, 0
        je      short   EndStrZ
        mov     ah, 06h
        int     21h
        inc     bp
        jmp     short   StrLp
EndStrZ:
        ret
OutStrZ ENDP

_TEXT   ENDS

        END     START           ;START からプログラムは実行される
```

常駐型のプログラム　TIMES.COM

“**TIMES.ASM**”がソース・プログラムです.

“**COM.BAT**”でアセンブル，リンクできます.

```
A>COM TIMES
```

MD-DOSではキー入力待ちのときには「**int　28h**」が発行されます（このことはNECのマニュアルには記載されていません）.

28hの割り込みベクタには通常，何もしないで元のルーチンに戻る「**IRET**」だけのルーチンが登録されています.

このことを利用して，時刻表示のプログラムとベクタを書き換えて，時刻表示を実現している簡単なプログラムです.

プログラムを実行すると，常駐判定ルーチン（**isStay**）に飛びます.

isStayでは，

①常駐していないと，**TIMES**を常駐して終了する.

②すでに**TIMES**が常駐していると常駐を解除（使用していたメモリを解放）して終了する.

のいずれかの処理を行ないます.

常駐しているかどうかの判定は28hの割り込みベクタに登録されているオフセット・アドレスと“**TIMES.COM**”中の割り込みプログラム（**NewInt28h**）のオフセット・アドレスが同じであるかどうかで判断しています.

TIMESを常駐して終了する

①元の28hの割り込みベクタを保存する.
②新しい28hの割り込みベクタ（時刻表示ルーチン）を登録する.
③読み込まれた**TIMES.COM**の環境エリアを解放する.
④常駐サイズを指定して，**TIMES.COM**を常駐終了する.

プログラムの先頭から常駐するので，

常駐部　→前 非常駐部　→後

という構成です.

そのほか，DOSの多重割り込みを防ぐため，**inDOS**フラグを監視しますが，そのためにあらかじめ**inDOS**フラグのアドレスを取得しておきます.

TIMESを解放して終了する

①元の28hの割り込みベクタを復元する.
②常駐していた**TIMES.COM**（メモリ）を解放する.
③**TIMES.COM**を終了する.

時刻を表示する

①使うレジスタを**push**します.
②**inDOS**フラグを見て1以外ならば何もせずに終わります.
MS-DOSでは割り込みルーチンの初めで**inDOS**フラグは+1され，その終わりで-1されています.
割り込みの中で割り込みが発生するということが起こるので，多重に割り込みが発生するとスタックが足りなくなるなどのトラブルが発生します.
したがって，**inDOS**フラグの値を見て，
0であれば割り込み中でない
1であれば割り込み中
2であれば割り込み中でもう1度割り込みが発生している
というように判断できます.
③256回に1回だけ表示するためのカウンタ
④時刻データを取得する
⑤時刻データを文字に変換する
⑥テキストVRAMに直接データを書く
この部分はPC-9801のテキストVRAMにデータを直接書き込んでいますので，他機種ではアドレス，アトリビュートの形式を変更する必要があります.
⑦**push**したレジスタをpopして，本来のベクタにジャンプしています.

```
REPLACE_INT     equ     28h             ;置き換える割り込み番号
PSPEnv          equ     0002ch          ;pspから環境変数のセグメントがわかる
DISPLAY_POSI    equ     70*2            ;PC-9801のテキストVRAMのオフセット

_TEXT   SEGMENT BYTE PUBLIC 'CODE'
        ASSUME  CS:_TEXT, DS:_TEXT

        ORG     100h

START   PROC    NEAR
        jmp     near ptr isStay
START   ENDP

NewInt28h       PROC    FAR
        push    ds
        push    es
        push    bx
        push    ax
        mov     bx, cs: [InDosFlagOff]  ;inDOSフラグのチェック
        mov     ax, cs: [InDosFlagSeg]
        mov     es, ax
        mov     al, es:[bx]
        cmp     al, 1                   ;1以外であれば何もしない
        jne     IntRet
        ;
        call    DisplayTime             ;時刻表示ルーチン
IntRet:
        pop     ax
        pop     bx
        pop     es
        pop     ds
        jmp     dword ptr       cs:[OldVectOff]

NewInt28h       ENDP

;-------------------------------------------------------
;       時刻表示
;-------------------------------------------------------
DisplayTime     PROC    NEAR
        mov     ax, cs
        mov     ds, ax          ; DSをCSに合わす

        mov     al, ds:[DspCount]       ;256回に1回表示する
        dec     al
        mov     ds:[DspCount], al
        jz      DspTimeMain
        ret
DspTimeMain:
        mov     ah, 2ch
        int     21h             ; ch=time cl=min dh=sec
        mov     bp, offset DspData
        mov     al, ch
        call    near ptr SetBcd
        inc     bp              ; :

        mov     al, cl          ;分
        call    near ptr SetBcd
        inc     bp              ; :
        mov     al, dh          ;秒
        call    near ptr SetBcd

        mov     bp, offset DspData
        mov     ax, 0a000h      ;PC-9801テキストVRAMのセグメント
        mov     es, ax
        mov     di, 0
DsptimeLp:
        mov     al, ds:[bp]
        cmp     al, 0
        je      DspTimeEnd
;       文字は1文字＝2バイト、アトリビュートは書き込まない
;-------------------------------------------------------
        mov     Byte ptr es:[di+DISPLAY_POSI], al
        inc     di
        mov     Byte ptr es:[di+DISPLAY_POSI], 0
        inc     di
        inc     bp
        jmp     short   DsptimeLp
DspTimeEnd:
        ret
DspData         db      '12:15:30',0
DisplayTime     ENDP
```

```
;-------------------------------------------------------
;       AL -> (BCD-ASCII) -> [BP]
;       BP: BCD値をセットするアドレス
;       AL: 16進データ
;-------------------------------------------------------
SetBcd  PROC    NEAR
        mov     ah, 0
SetBcdLp:
        cmp     al, 10
        jb      short   SetBcdDat
        sub     al, 10
        inc     ah
        jmp     short   SetBcdLp
SetBcdDat:
        add     ah, '0'
        mov     ds:[bp], ah
        inc     bp
        add     al, '0'
        mov     ds:[bp], al
        inc     bp
        ret
SetBcd  ENDP

DspCount        db      1
InDosFlagOff    dw      ?
InDosFlagSeg    dw      ?
OldVectOff      dw      ?
OldVectSeg      dw      ?


;-------------------------------------------------------------
;       ここから後は常駐しない
;-------------------------------------------------------------
isStay  PROC    NEAR
StayLast:
        call    near ptr MiGetVect      ; --> ES:BX
        mov     ax, offset      NewInt28h
        cmp     bx, ax
        je      reMove
Stay:
        call    near ptr GetInDosFlag
        call    near ptr MiGetVect      ; --> ES:BX
        mov     ds:[OldVectOff], bx
        mov     ds:[OldVectSeg], es

        mov     dx, offset NewInt28h
        mov     ax, 2500h+REPLACE_INT
        int     21h     ; ds = seg

        mov     es, ds:[PSPEnv] ; Environment Segment address
        mov     ax, 4900h
        int     21h

        mov     ax, offset StayLast
        mov     cl, 4
        shr     ax, cl
        inc     ax
        mov     dx, ax
        mov     ax, 3100h
        int     21h
reMove:
;       古いベクタを復活
        call    near ptr MiGetVect      ; --> ES:BX
        mov     bx, offset OldVectOff
        mov     ax, es:[bx]             ; old offset
        mov     dx, ax
        mov     ax, es:[bx+2]           ; old segment
        mov     ds, ax
        mov     ax, 2500h+REPLACE_INT
        int     21h                     ; DS:DX
        ;
        mov     ah, 49h         ; seg = ES
        int     21h             ; free regident
        ;
        mov     ax, 4c00h
        int     21h
isStay  ENDP
```

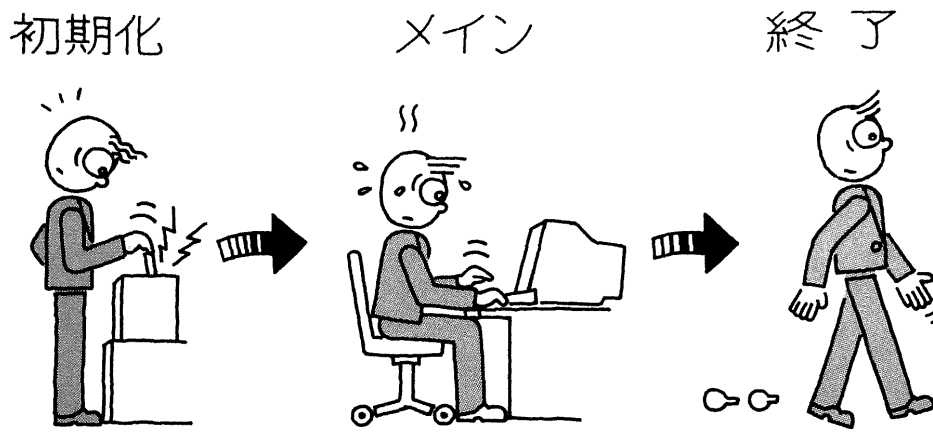
```
MiGetVect       PROC    NEAR
        mov     ax, 3500h + REPLACE_INT
        int     21h             ; --> ES:BX
        ret
MiGetVect       ENDP

;-------------------------------------------------------
;       inDOSフラグのアドレスを得る
;       多重割り込みを防ぐために使用
;-------------------------------------------------------
GetInDosFlag    PROC    NEAR
        push    es
        push    bx
        push    ax
        mov     ah, 34h
        int     21h
        mov     [InDosFlagSeg], es
        mov     [InDosFlagOff], bx
        pop     ax
        pop     bx
        pop     es
        ret
GetInDosFlag    ENDP


_TEXT   ENDS
        END     START
```

CBW CWD ESC
IN JCXZ
HF LDS LEA
CK LODS LOOP
OP NOT OUT
SH PUSHF REP
G STOS WAIT

第6部　プログラムの実行開始と終了

プログラムの実行開始時点のさまざまな初期化や終了時の処理はだいたい同じなので，定型化してみましょう．

XCHG XLAT
DD CMP CMPS
BB SCAS SUB
OR SAR SHL
TEST XOR
INC
CR ROL ROR



プログラムの初期化部分

初期化部分ですることは，セグメント・レジスタの初期化とそのプログラム固有のデータの初期化などです．

それぞれのプログラム固有の部分については述べませんが，グローバル，ローカルに分けて考えるなど意識的な操作がトラブルを避ける方法でしょう．

《COMモデル》

COMモデルでは基本的に1つのセグメントのみを扱い，プログラムが動作を始めた時点でCS，DS，SS，ESはそのプログラムのセグメントを指しているので特別な処理は必要ありません．

CS=DS=SS=ES = PSP
IP=100h

《EXEモデル》

EXEモデルでは，

END　　実行開始プロシージャ名

で指定されたプロシージャから実行が始まります．

このとき各セグメント・レジスタの値はEXEヘッダに記録されている値（リロケーション・データ）をもとにセットされます．

CS:IP ←EXEヘッダの情報
SS:SP ←EXEヘッダの情報
DS=ES= PSP

EXEファイルではデータ・セグメントはコード・セグメントとは別なのが普通です．そのため，「**DS=ES=CS**」となっていると都合が悪いのです．

データ・セグメントの値は，プログラムの比較的初めのあたりにある，

```
mov     ax, seg DATA_SEG
mov     ds, ax
```

のようなコードで設定します．

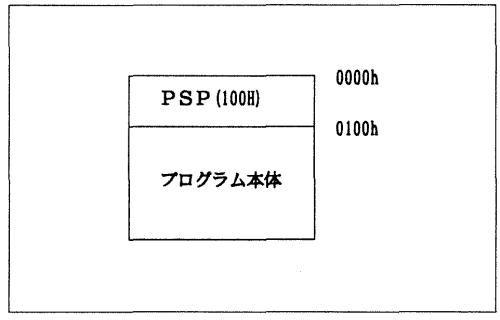
「比較的初めのあたり」と書いたのは，コマンドライン・オプションなどの情報をPSP経由で受けとるため，その処理を先にする必要があるからです．

したがって，初期化部分の流れは，
①コマンドライン・オプションの読み取り
②DSにデータ・セグメントをセット
となります．

コマンドライン・オプションなどの受け渡し

コマンドライン・オプションはDOSがプログラムを起動するときにPSPのアドレス80h以降にコピーされます．

MS-DOSが実行するプログラムをメモリに読み込み，読み込まれたプログラムに制御を移す(プログラムが実行される)とき，DOSはPSPを作り，その中に必要な情報をセットします．



PSPのサイズは100h(256バイト)で多くの情報が含まれていますが，プログラムを書くうえで最初に必要になる情報がコマンドライン・オプションの受け渡しです．

サンプルの「**CLINE1.ASM**」は，PSP領域をダンプ(16進表示)するプログラムです．
先頭の命令で**PSP**の先頭アドレスを**SI**レジスタにセットしています．

```
        mov     si, 0                   ; PSPの先頭を指す
```

COM形式のプログラムでは最初(**CLINE1.EXE**が動き始めたとき)**IP**は**100h**で，**DS=CS=PSP**のセグメントになっています(DOSがそうする)．
したがって**PSP**の先頭アドレスは「**PSP**のセグメント:0000」なので**SI=0**とすれば**DS**をあえてセットしないでも以後**DS:SI**で**PSP**を参照することができます．
EXE形式のプログラムではその前に前述の，

```
        MOV     AX, SEG DATA_SEG
        MOV     DS, AX
```

を記述します．

簡単なプログラムなので構造，関数などはプログラム中のコメントを参照してください．

では実際に「**CLINE1.EXE**」を実行して**PSP**領域を表示してみましょう．コマンド・ラインから，

```
A>CLINE1 OPTION
```

とタイプしてみてください．次のように表示されます．

画面1　CLINE1.COMの実行結果

```
A>CLINE1 OPTION
CD 20 00 A0 00 9A F0 FE 1D F0 2F 01 D0 20 3C 01
D0 20 EB 04 D0 20 D0 20 01 01 01 00 02 FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF BD 21 E0 FF
D0 21 14 00 18 00 D0 21 FF FF FF FF 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
CD 21 CB 00 00 00 00 00 00 00 00 00 4F 50 54
49 4F 4E 20 20 20 20 20 00 00 00 00 00 20 20 20
20 20 20 20 20 20 20 20 00 00 00 00 00 00 00 00
07 20 4F 50 54 49 4F 4E 0D 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A>
```

9行目のオフセット80hのところを見てください．

先頭に文字数(07)，次がスペース(20)，それ以降に文字列(OPTION)が入っています．

文字数には先頭のスペースも含まれます．

```
07 20 4F 50 54 49 4F 4E 0D 00 00 00 00 00 00 00
↑  ↑  ↑  ↑  ↑  ↑  ↑  ↑
      O  P  T  I  O  N
```

コマンド名とオプションを区切る空白．この後にオプションの文字列

コマンドライン・オプションの文字数

コマンドライン・オプションにより**PSP**領域のデータがどのように変わるかを，オプションを変えて実行して確認してください．

要約：

PSPのオフセット**80h**にはコマンドライン･オプションの文字数がセットされ，**81h**以降にオプションの文字列がセットされます．

PSPのサイズは**100h**（256）バイトです．オプションに使える領域は80hからなので，**PSP**領域の半分の**128**バイトが使えます．

コマンド名とオプションを区切る空白もコピーされるので，最大で126文字以下のオプションが使用できることになります．

言葉で説明するより，実際にプログラムを使って説明するほうが理解しやすいと思いますので，コマンドライン・オプションに関するサンプル・プログラムをあと２つ付属しました．

CLINE2.COMと**CLINE3.COM**です．ソースはそれぞれ**CLINE2.ASM**と**CLINE3.ASM**です．ソースを見ながら色々試してみると理解しやすいと思います．

簡単に動作を説明します．

CLINE2

オプション部分だけを［］に囲んで表示します．

CLINE3

複数のオプションを分離し，［］に囲んで表示します．

このプログラムは実際に使われているソフトに近い仕様なので参考になるかもしれません．

```
A>CLINE3 -OPTION -OPTION2
A>CLINE3 -OPTION -OPTION2/OPTION3
```

など試してみてください．

オプション始まりのマークには「-」（ハイフン）を使います．

「-」以外で始まる文字列はエラーにします（実際のプログラムではファイル名に解釈したりします）．

大文字，小文字の処理はしていません．

オプションとオプションの間はスペースで区切っても，区切らなくてもかまいません．

ただし，実際はソフトによってコマンドライン・オプションの指定のやりかた（文法）がかなり違います．

大きく分類すると，

①文字列か1文字か

例：　有名なものでは

文字列型…マイクロソフトのコンパイラやアセンブラなど

/NOI[GNORECASE] /ST[ACK] など（[]内は省略可）

1文字型…ボーランドのコンパイラなど

-v -s -ml など

②大文字，小文字を区別する，しない．

③オプション始まりのマーク

「-」，「/」どちらか，または両方．

通常それ以外はファイル名などに解釈します．

④オプションを続けて記述できる

などです．

初期化の最小公倍数を考えてみましょう．それにはC言語のスタートアップ・ルーチンを見てみるとよく分かります．

最近のCコンパイラにはスタートアップ・ルーチンのソース・リストが付属していることがあります．ボーランドの**Borland C++**，**Turbo C**などには“**C0.ASM**”というファイルが含まれています（圧縮された形で付属しています）．

マクロ，if文がふんだんに使われていて，慣れるまではかなり見づらいかもしれませんが，持っている人は一度見てみるとよいでしょう．

このスタートアップ・ルーチンでは，

①データのセグメントを広域変数にセーブ
②DOSのバージョンを調べて広域変数にセーブ
③PSPのアドレス（セグメント値）を広域変数にセーブ
④割り込みベクタのセーブ
⑤割り込みベクタの変更
⑥メモリの空き容量のチェック
⑦コマンドラインの解析
⑧DSのセット

などをしています．あくまでも最小公倍数なので，普通はここまでは必要ないでしょう．

最近は常駐型のプログラムもC言語で書くことが多くなっています．

この場合，常駐部だけアセンブラで書いて，その他の部分はC言語で書くスタイルが多いようです．

このとき注意しなければならないのは「**⑤割り込みベクタの変更**」の部分です．常駐型なので当然C言語の終了ルーチンを使わずにプログラムを終了します．

つまり，割り込みベクタを元に戻さずプログラムは終了してしまうのです．

スタートアップ・ルーチンを変更して，最低限「割り込みベクタの変更」部分を削除する必要があります．

終了部分

終了

プログラムを終了するには何種類かの方法があります.

① **int 20h**
② **int 21h AH=00h**
③ **int 21h AH=4Ch**

「プログラムの実行結果により違う処理をする」記述をバッチ・ファイルなどでよく見かけます.

これは，プログラムが終了するときにDOSに数値（リターン・コード）を渡しているから，このような判断が可能になっています.

DOSに数値を渡して終了できるのは③の方法です．ALにリターン・コードを入れて実行します．したがって0～255（FFh）までの数値を渡すことができます.

多くのプログラムは，プログラムが正常に終了したときは0を返し，それ以外では1を返しています.

プログラムによってはエラーの種類により割り当てられたコードを返し，エラーコードを公表してあるソフトもあります.

あえて古い終了方法を使用する必要はないのでこの③の方法を使うのがよいでしょう.

```
        mov     ax, 4c00h
        int     21h
```

リターン・コードの利用

例：プログラムを実行し，その結果が1以上であれば「ラベル」へジャンプするバッチ・ファイルを考えてみます.

```
プログラム
if errorlevel 1 goto ラベル
  ～～～
:ラベル
  ～～～
```

ここで，よく誤解するのは，

「**if errorlevel 1**」を「= 1」と考えることです.

「≧ 1 」が正解です.

したがって，0～3のリターンコードを返すプログラムでは，

```
if errorlevel 3 goto label3
if errorlevel 2 goto label2
if errorlevel 1 goto label1
:label0
  ～～～
:label1
  ～～～
:label2
  ～～～
:label3
```

のように大きな数値から評価する必要があります．

常駐終了

プログラムを常駐終了するのにも2種類の方法があります．

```
① int 27h
② int 21h  AH=31h
```

①の方法には，

64K以上は常駐できない．

リターン・コードを返せない．

などの欠点があります．

そのため，②の「**int 21h AH=31h**」を使うべきでしょう．

```
mov     dx, 常駐サイズ（パラグラフ）
mov     ax, 3100h
int     21h
```

CBW CWD ESC
IN JCXZ
HF LDS LEA
CK LODS LOOP
OP NOT OUT
SH PUSHF REP
EG STOS WAIT

第7部　サンプル・プログラム解説

付属のディスクにはZZ386の逆アセンブル用のサンプル・ターゲットとして，少しは実用的なプログラムが3つ入っています．

以下に簡単に内容とアルゴリズムなどを説明します．

詳細についてはソース・リストを見てください．

XCHG XLAT
DD CMP CMPS
BB SCAS SUB
OR SAR SHL
TEST XOR
INC
CR ROL ROR

CAL.COM　今日の日付を日本語で表示する.

《使い方》

実行すると,

```
A>CAL
        １９９２年１１月１８日 水曜日
```

というように表示されます.

「**AH=2Ah　int 21h**」を使ってコンピュータのシステム時間を取得しています.

16進数から10進数への変換や日本語全角文字への変換ルーチンなどが参考になると思います.

DSIZE.COM　ドライブの残り容量などを表示する.

《使い方》

```
A>DSIZE <ドライブ>
```

セクタ・サイズ，クラスタ・サイズ，ドライブの容量などが表示されます.

《使用例》

```
A>DSIZE C
    ﾄﾞﾗｲﾌﾞ = C:
    ﾊﾞｲﾄ/ｾｸﾀ 512. ｾｸﾀ/ｸﾗｽﾀ 8. 残りｸﾗｽﾀ 2502. 全ｸﾗｽﾀ 59671.
    残り容量 10,248,192 ﾊﾞｲﾄ 全容量 244,412,416 ﾊﾞｲﾄ
```

「**AH=36h int 21h**」でドライブの空き情報を得ています.

数字の表示ルーチンなどが参考になると思います.

FS.EXE　カレント・ドライブ全体から指定されたファイル名のファイルを検索する

ファイル名にはワイルド・カードが使えます.

ハードディスクなどでサブディレクトリが混みあってくると，「あのファイルはどこだっけ？」などということがよくあります. そういうときに重宝するプログラムです.

《使い方》

```
A>FS ファイル名
```

《例》

```
A>FS FS.ASM
A>FS *.ASM
A>FS F*.EXE
A>FS FS.*
```

《使用例》

```
A>FS FS.*                                    ←コマンドライン
検索中です=[FS.*]                            ←検索ファイルを表示
¥TOOL¥FS.EXE                                 ↓見つかったファイルを順次表示
¥TOOL¥FS.EXE
¥TOOL¥FS.MAP
¥TOOL¥FS.OBJ
¥TOOL¥FS.LST
¥TOOL¥FS.BAK
¥TOOL¥FS.ASM
見つかったファイル数=7 検索したサブ･ディレクトリ数=461
```

《プログラムの概要》

ファイルの検索には次の2つのファンクションを使います.

int 21h AH=4E	最初のファイル検索
int 21h AH=4F	次のファイル検索

「最初のファイル検索」はパラメータとして,

CX=属性（ファイルは00H, サブディレクトリは10H）
DS:DX=ファイル名のアドレス

をセットして実行します.
ファイルが見つからないときはキャリーフラグがセットされます.

続いて同じパス(ファイル名)をサーチするときは「次のファイル検索」を使います.
このときは属性(**CX**)とファイル名(**DS:DX**)は必要ありません.

通常, この2つのファンクション「最初のファイル検索」と「次のファイル検索」はセットで使われます.
そのため, 次のような順序では使えません.

《不可能》

```
最初のファイル検索 [ABC.*]
最初のファイル検索 [XYZ.*]
次のファイル検索 [ABC.*]
次のファイル検索 [XYZ.*]
```

《この順序が正しい》

```
最初のファイル検索 [ABC.*]
次のファイル検索 [ABC.*]
...
最初のファイル検索 [XYZ.*]
次のファイル検索 [XYZ.*]
...
```

ＦＳの検索ルーチンのアルゴリズム

①パス(最初は¥)とファイル名(ABC.*など)を受け取る.

②「パス＋ファイル名」のファイルを検索する(最初は¥ABC.*).
　見つかったファイルを表示する.

③「パス＋*.*」のサブディレクトリ名(最初は¥*.*)をすべて検索する.
　見つかったサブディレクトリはすべて記憶しておく.

④「見つかったサブディレクトリ名(パスとして）＋ファイル名(ABC.*など)」をパラメータとして「検索ルーチン」をコールする(再帰コール).

《動作例》

最初に「ルートのパス」と「検索ファイル名」をパラメータとして，「検索ルーチン」を呼ぶ

[¥] + 検索ファイル名＝[ABC.*]

↓

「検索ルーチン」

①検索ファイル名の合成

[¥] + [ABC.*] = [¥ABC.*]

②¥ABC.* を検索する

¥ABC.EXE　　が見つかった → 表示

¥ABC.ASM　　が見つかった → 表示

¥ABC.* の検索終了

③¥*.* のサブディレクトリを検索する

¥TOOL　　が見つかった → 記憶する

¥DOS　　が見つかった → 記憶する

¥EDIT　　が見つかった → 記憶する

④記憶しておいたサブディレクトリと検索ファイル名をパラメータにして「検索ルーチン」を呼ぶ(再帰コール)

[¥TOOL] + [ABC.*]

↓

「検索ルーチン」

「検索ルーチン」の終了

IN JCXZ
HF LDS LEA
CK LODS LOOP
OP NOT OUT
SH PUSHF REP
G STOS WAIT

MS-DOS Version 5.0
ファンクション・リクエスト(INT21h)一覧

いわゆるファンクション・リクエスト(INT 21h)はAHレジスタにファンクション番号を入れ，必要であればALレジスタにサブ・ファンクション番号，その他のレジスタに必要であればパラメータ(データ)を入れてINT 21hを発行します．

使い方の詳細，レジスタの保存，破壊などの情報はMS-DOSのプログラマーズ・ガイドなどを参照してください．

ZF=ゼロ・フラグ，CF=キャリー・フラグ
IN: AH以外にセットするレジスタ
OUT: 割り込みから戻ったときに値がセットされているレジスタ

XCHG XLAT
DD CMP CMPS
BB SCAS SUB
OR SAR SHL
TEST XOR

00h プログラムの終了

IN: **CS** =**PSP**のセグメント・アドレス
OUT: なし
備考: ファンクション**4Ch**の使用が望ましい.

01h 文字入力（エコーあり），文字入力まで待つ

IN: なし
OUT: **AL** =入力された文字

02h 文字出力

IN: **DL** 出力する文字コード
OUT: なし

03h 補助入力（RS232C）

IN: なし
OUT: **AL** =補助装置から入力された文字

04h 補助出力（RS232C）

IN: **DL** =補助装置に出力する文字コード
OUT: なし

05h 文字のプリンタ出力

IN: **DL** =プリンタに出力する文字
OUT: なし

06h 直接コンソール入出力

IN: **DL** =**FFh**（入力）
=**FFh**以外（出力）
OUT: （**DL=FFh**の場合）
ZF =1（文字入力なし）
=0（文字入力あり）
AL =文字
（**DL≠FFh**の場合）
なし
備考: **DL**が**FFh**のときは文字入力を待たない
DLが**FFh**以外のときは**DL**のキャラクタを出力

07h 直接コンソール文字入力

IN: なし
OUT: **AL** =キーボードから入力された文字
備考: 文字入力まで待つ

08h 文字入力（エコーなし）

IN: なし

OUT: **AL** =キーボードから入力された文字

備考: 入力された文字は画面に表示されない.

09h 文字列の表示

IN: **DS:DX** =文字列の先頭アドレス

OUT: なし

備考: 文字列は**'$'**で終る，**'$'**は出力されない．例 **'STRING','$'**

0Ah バッファドキーボード入力

IN: **DS:DX** =入力バッファのアドレス

OUT: なし

備考: キャリッジ・リターンか設定された最大文字数までをキーボードから読み取る

入力バッファの形式

オフセット	内　容
0	読み込む最大文字数（1～255）
1	読み込んだ文字数がセットされる
2～	読み込んだ文字のデータ

0Bh キーボード・ステータスの検査

IN: なし

OUT: **AL** **=FFh**（バッファ内に文字が入っている）
=00h（バッファ内に文字が入っていない）

0Ch バッファを空にしてキーボード入力

IN: **AL** **=01h,06h,07h,08h,0Ah**
（対応するファンクション番号）

OUT: 備考を参照

備考: **AL**が上に挙げた値の場合はキーボード・バッファをクリアして**AL**の値のファンクションを実行する．それ以外の値の場合はバッファをクリアする．バッファのクリアだけを実行した場合の戻り値は**AL=00h**，ファンクションを実行した場合はそれぞれのファンクションを参照.

0Dh ディスクのリセット

IN: なし

OUT: なし

備考: すべてのファイル・バッファの内容をディスクに書き出し，ファイル・バッファを空にする．ディレクトリエントリの更新のためには更新されたファイルをクローズする.

0Eh ディスクの選択

IN: **DL** =ドライブ番号（**00h=A:,01h=B:,**…）

OUT: **AL** =論理ドライブの数

備考: 返される論理ドライブの数は**CONFIG.SYS**の**LASTDRIVE**指定で設定された数で，実際のドライブ数ではないので注意.

0Fh FCBを使ったファイルのオープン

IN: **DS:DX** =オープンするFCBのアドレス

OUT: **AL** =**00h**（正常にオープンした）

=**FFh**（オープンできなかった，ファイルが存在しない）

備考: ファンクション**3Dh**の使用が望ましい.

10h FCBを使ったファイルのクローズ

IN: **DS:DX** =オープンされているFCBのアドレス

OUT: **AL** =**00h**（正常にクローズした）

=**FFh**（クローズできなかった，ファイルが存在しない）

備考: ファンクション**3Eh**の使用が望ましい.

11h FCBを使った最初のファイルの検索

IN: **DS:DX** =オープンされていないFCBのアドレス

OUT: **AL** =**00h**（ファイルが見つかった）

=**FFh**（ファイルは見つからない）

備考: ファンクション**4Eh**の使用が望ましい.

12h FCBを使った次のファイルの検索

IN: **DS:DX** =オープンされていないFCBのアドレス

OUT: **AL** =**00h**（ファイルが見つかった）

=**FFh**（ファイルは見つからない）

備考: ファンクション**4Fh**の使用が望ましい.

13h FCBを使ったファイルの削除

IN: **DS:DX** =オープンされていないFCBのアドレス

OUT: **AL** =**00h**（削除した）

=**FFh**（削除できなかった，ファイルが存在しない）

備考: ファンクション**41h**の使用が望ましい.

14h FCBを使ったシーケンシャルな読み出し

IN: **DS:DX** =オープンされているFCBのアドレス

OUT: **AL** **=00h**（正常な読み込み）
=01h（EOF，データなし）
=02h（DTAで示されるバッファが小さすぎる）
=03h（レコードの一部分にEOFまでのデータが入り，残りがゼロで埋められた）

備考: ファンクション**3Fh**の使用が望ましい．

15h FCBを使ったシーケンシャルな書き込み

IN: **DS:DX** =オープンされているFCBのアドレス

OUT: **AL** **=00h**（正常な書き込み）
=01h（ディスクに空き領域がない）
=02h（DTAで示されるバッファが小さすぎる）

備考: ファンクション**40h**の使用が望ましい．

16h FCBを使ったファイルの作成

IN: **DS:DX** =オープンされていないFCBのアドレス

OUT: **AL** **=00h**（作成できた）
=FFh（作成できなかった）

備考: 既に存在するファイルに体して実行した場合，上書きされる．
用途によって，ファンクション**3Ch,5Ah,5Bh**のどれかの使用が望ましい．

17h FCBを使ったファイル名の変更

IN: **DS:DX** =修正されたFCBのアドレス

OUT: **AL** **=00h**（ファイル名を変更できた）
=FFh（変更できなかった）

備考: FCBに旧名と新名をあらかじめセットしておく（非標準フォーマット）

FCBの形式

オフセット	内　容
00h	ドライブ番号
01～08h	旧ファイル名 （未使用部分はスペースで埋める）
09～0Bh	旧拡張子名 （未使用部分はスペースで埋める）
0C～10h	0で埋める
11～18h	新ファイル名 （未使用部分はスペースで埋める）
19～1Bh	新拡張子名 （未使用部分はスペースで埋める）
1C～24h	0で埋める

＊拡張FCBを使うとサブディレクトリも変更できる

19h カレント・ドライブ番号の取得

IN: なし

OUT: **AL** =カレントドライブ番号（**00h=A:,01h=B:,**…）

1Ah DTA（ディスク・トランスファー・アドレス）の設定

IN: **DS:DX** =DTAのアドレス

OUT: なし

備考: 設定しないときはPSPの80hからの128バイトが使われる

1Bh カレント・ドライブのデータの取得

IN: なし

OUT: **AL** =**FFh**（データが得られない）
=クラスタ当たりのセクタ数
CX =セクタ当たりのバイト数
DX =ドライブ当たりのクラスタ数
DS:BX =**FAT-ID**のアドレス

1Ch ドライブ・データの取得

IN: **DL** =ドライブ番号（**00h**=カレント,**01h=A:,02h=B:,**…）

OUT: **AL** =**FFh**（データが得られない）
=クラスタ当たりのセクタ数
CX =セクタ当たりのバイト数
DX =ドライブ当たりのクラスタ数
DS:BX =**FAT-ID**のアドレス

21h FCBを使ったランダムな読み出し

IN: **DS:DX** =オープンされているFCBのアドレス

OUT: **AL** =**00h**（正常な読み込み）
=**01h**（EOF，データなし）
=**02h**（DTAで示されるバッファが小さすぎる）
=**03h**（EOF，レコードの一部分）

備考: ファンクション**3Fh**の使用が望ましい

22h FCBを使ったランダムな書き込み

IN: **DS:DX** =オープンされているFCBのアドレス

OUT: **AL** =**00h**（正常な書き込み）
=**01h**（ディスクに空き領域がない）
=**02h**（DTAで示されるバッファが小さすぎる）

備考: ファンクション**40h**の使用が望ましい

23h FCBを使ったファイルの大きさの取得

IN: **DS:DX** =オープンされていないFCBのアドレス

OUT: OUT: **AL** **=00h**（ファイルが存在する）

=FFh（ファイルが存在しない）

備考: ファンクション**42h**の使用が望ましい

24h 相対レコードの設定

IN: **DS:DX** =オープンされているFCBのアドレス

OUT: **AL** =0（必ず0になる）

備考: ファンクション**22h,23h,27h,28h**を実行する前にコールする.

ファンクション**42h**の使用が望ましい

25h 割り込みベクタの設定

IN: **AL** =割り込み番号

DS:DX =割り込み処理ルーチンのアドレス

OUT: なし

26h 新しいPSPの作成

IN: **DX** =新しいPSPのセグメントアドレス

OUT: なし

備考: **AL=00H**，ファンクション**4BH**の使用が望ましい

27h FCBを使ったランダムなブロックの読み出し

IN: **DS:DX** =オープンされているFCBのアドレス

CX =読み出すレコード数

OUT: **AL** **=00h**（正常な読み込み）

=01h（EOF，空レコード）

=02h（DTAで示されるバッファが小さすぎる）

=03h（EOF，レコードの一部分）

CX =読み取ったレコード数

備考: ファンクション**3Fh**の使用が望ましい

28h FCBを使ったランダムなブロックの書き込み

IN: **DS:DX** =オープンされているFCBのアドレス

CX **00h**（ファイルサイズフィールドの設定）

=書き込むレコード数（CX≠00h）

OUT: **AL** **=00h**（正常な書き込み）

=01h（ディスクに空き領域がない）

=02h（DTAで示されるバッファが小さすぎる）

CX =書き込まれたレコード数

備考: ファンクション**40h**の使用が望ましい

29h ファイル名の解析

IN: **AL** =解析の制御コード
DS:SI =解析する文字列
ES:DI =オープンされていないFCBのアドレス

OUT: **AL** =**00h**（ワイルドカードが使用されていない）
=**01h**（ワイルドカードが使用されている）
=**FFh**（ドライブ文字が無効）
DS:SI =解析した文字列の直後のアドレス
ES:DI =オープンされていないFCBのアドレス

2Ah 日付（システム日付）の取得

IN: なし

OUT: **CX** =年（1980～2099）
DH =月（1～12）
DL =日（1～31）
AL =曜日（**00h**=日，**01h**=月，…**06h**=土）

2Bh 日付の設定

IN: **CX** =年（1980～2099）
DH =月（1～12）
DL =日（1～31）

OUT: **AL** =**00h**（日付は設定された）
=**FFh**（日付は設定できなかった）

2Ch 時刻の取得（システム時刻）

IN: なし

OUT: **CH** =時（0～23）
CL =分（0～59）
DH =秒（0～59）
DL =100分の1秒（0～99），機種により00がセットされる

2Dh 時刻の取得（システム時刻）

IN: **CH** =時（0～23）
CL =分（0～59）
DH =秒（0～59）
DL =00h

OUT: **AL** =00h（時刻はセットされた）
=FFh（時刻がセットできなかった）

2Eh ベリファイ・フラグの設定

IN: **AL** =00h（ベリファイをオンにする）
=01h（ベリファイをオフにする）
DL =00h（MS-DOS Ver1.x,2.xのみ）*
OUT: なし
備考: DOS内部のベリファイ・フラグをオン/オフする.

2Fh DTAの取得

IN: なし
OUT: **ES** =現在のDTAのセグメント
BX =現在のDTAのオフセット

30h MS-DOSバージョン番号の取得

IN: **AL** **=00h**（Ver.5で**BH**にOEMシリアルを返す場合）
=01h（Ver.5で**BH**にバージョンフラグを返す場合）
OUT: **AL** =バージョン番号の整数部（**X.YY**の**X**．0の場合Ver.2.0以前）
AH =バージョン番号の小数部（**X.YY**の**YY**）
BH =OEMシリアル番号（**00h,FFh**など．機種によって違う）
=バージョン・フラグ（**08h**ならDOSはROM上にある）
BL:CX =ユーザーシリアル番号（000000hなど．機種によって違う）

31h プロセスの常駐終了

IN: **AL** =終了コード
DX =常駐するメモリのサイズ（パラグラフ(16バイト)単位）
OUT: なし

33h システム値の取得および設定

Ctrl-Cのチェック・フラグ状態の取得

IN: **AL** **=00h**

OUT: **AL** **=FFh** 状態を読み出せない

=00h 正常に読み出せた

(以下**AL=00h**の場合)

DL **=00h**(状態はオフ)

=01h(状態はオン)

Ctrl-Cのチェック・フラグの設定

IN: **AL** **=01h**

DL **=00h**(オフにする)

=01h(オンにする)

OUT: **AL** **=00h**(セットできた)

=FFh(セットできない)

ブート・ドライブの取得 ＊Ver.5のみ

IN: **AL** **=05h**

OUT: **DL** =ドライブ番号(01h=A:,02h=B:,…)

MS-DOSバージョンの取得(SETVER対応) ＊Ver.5のみ

IN: **AL** **=06h**

OUT: **BL** =正確なバージョン番号の整数部(**X.YY**の**X**)

BH =正確なバージョン番号の小数部(**X.YY**の**YY**)

DL =下位の3ビットにリビジョン番号，ほかのビットは0

DH =MS-DOSのバージョン・フラグ

08h(MS-DOSはROMにある)

10h(MS-DOSはハイメモリにある)

34h inDOSフラグのアドレスを得る

IN: なし

OUT: **ES:BX** =inDOSフラグのアドレス.

備考: NECのマニュアルには記載なし

35h 割り込みベクタの取得

IN: **AL** =割り込み番号

OUT: **ES:BX** =割り込みルーチンのアドレス

36h ディスクの空き容量の取得

IN: **DL** =ドライブ番号（**00h**=カレント,**01h=A:**,**02h=B:**,…）

OUT: **AX** =**FFFFh**（ドライブ番号が無効）
=クラスタ当たりのセクタ数

BX =使用可能なクラスタ数（未使用）

CX =セクタ当たりのバイト数

DX =ドライブ当たりのクラスタ数

38h 国別情報の取得，設定

取得

IN: **AL** =情報を取得する国のカントリーコード．**FFh**以上は**BX**で指定．

BX =情報を取得する国のカントリーコード（**AL=FFh**のとき）

DS:DX =情報に対する34バイトのバッファのアドレス

OUT: **CF** =**0**（**DS:DX**で指定したバッファに情報が設定された）
=**1**（エラー）

備考: **AL**=の場合は現在カントリーコードの国の情報を取得する．

設定

IN: **DX** =**FFFFh**

AL =設定する国のカントリー・コード

BX =設定する国のカントリー・コード（AL=FFhのとき）

OUT: **CF** =**0**（エラーなし）
=**1**（エラー）

備考: 通常，カントリー・コードは国際電話のプリフィックス・コードと同じ．

39h ディレクトリの作成

IN: **DS:DX** =パス名を示すASCIIZ文字列のアドレス

BX =設定する国のカントリー・コード（AL=FFhのとき）

OUT: **CF** =**0**（エラーなし）
=**1**（エラー）

AX =エラーコード

備考: ASCIIZ文字列とは0（00h）で終わる文字列．

3AH ディレクトリの削除

IN: **DS:DX** =パス名を示すASCIIZ文字列のアドレス

OUT: **CF** =**0**（エラーなし）
=**1**（エラー）

AX =エラーコード

3Bh カレント・ディレクトリの変更

IN: **DS:DX** =パス名を示すASCIIZ文字列のアドレス

OUT: **CF** =**0**（エラーなし）

=**1**

AX =エラーコード

3Ch ハンドルを使ったファイルの作成

IN: **DS:DX** =パス名を示すASCIIZ文字列のアドレス

CX =ファイルの属性

OUT: **CF** =0（作成できた）

=1（エラー）

AX =ファイルハンドル番号（**CF=0**の場合）

=エラーコード（**CF=1**の場合）

3Dh ハンドルを使ったファイルのオープン

IN: **AL** =ファイルアクセスコード

DS:DX =パス名を示すASCIIZ文字列のアドレス

OUT: **CF** =0（オープンできた）

=1（エラー）

AX =ファイルハンドル番号（**CF=0**の場合）

=エラーコード（**CF=1**の場合）

3Eh ハンドルを使ったファイルのクローズ

IN: **BX** =ファイルハンドル番号

OUT: **CF** =0（エラーなし）

=1（エラー）

AX =エラーコード（**CF=1**の場合）

3Fh ファイル，デバイスの読み出し

IN: **BX** =ファイルハンドル番号

CX =読み込むバイト数

DS:DX =バッファのアドレス

OUT: **CF** =0（正常）

=1（エラー）

AX =読み込まれたバイト数（**CF=0**の場合）

=エラーコード（**CF=1**の場合）

40h ファイル，デバイスへの書き込み

IN: **BX** =ファイルハンドル番号
CX =読み込むバイト数
DS:DX =バッファのアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =書き込まれたバイト数（**CF=0**の場合）
=エラーコード（**CF=1**の場合）

41h ファイルの削除（ワイルド・カードは使えない）

IN: **DS:DX** =パス名を示すASCIIZ文字列のアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =エラーコード（**CF=1**の場合）

備考: リードオンリー・ファイルは削除できない.

42h ファイル・ポインタの移動

IN: **AL** =移動方法（**00**=先頭から,**01**=現在の位置から,**02**=最後から）
BX =ファイルハンドル番号
CX:DX =移動するバイト数（**CX=HIGH, DX=LOW**）

OUT: **CF** =**0**（正常）
=**1**（エラー）
DX:AX =新規のファイル・ポインタの位置（**CF=0**の場合）
AX =エラーコード

43h ファイル属性の取得，設定

取得

IN: **DS:DX** =パス名を示すASCIIZ文字列のアドレス

OUT: **CF** =0（正常）

=1

AX =エラーコード（**CF=1**の場合）

CX =属性（**CF=0**の場合）

（CXのビットの意味）

ビット	意　味
0	リードオンリー
1	不可視ファイル
2	システムファイル
5	アーカイブ

設定

IN: **DS:DX** =パス名を示すASCIIZ文字列のアドレス

CX =設定する属性

OUT: **CF** =0（正常）

=1（エラー）

AX =エラーコード

44h IOCTLファンクション

IOCTL:デバイス・データの取得

IN: **AL** =**00h**

BX =ファイルハンドル番号

OUT: **CF** =**0**（正常）

=**1**（エラー）

DX =デバイスデータ

AX =エラーコード

IOCTL:デバイス・データの設定

IN: **AL** =**01h**

BX =ファイルハンドル番号

DX =デバイスデータ

OUT: **CF** =**0**（正常）

=**1**（エラー）

AX =エラーコード

IOCTL:キャラクタ・デバイスからコントロール・データを受け取る

IN: **AL** =**02h**
BX =ファイルハンドル番号
CX =コントロール・データのバイト数
DS:DX =バッファのアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =受けとったバイト数（**CF=0**の場合）
=エラーコード（**CF=1**の場合）

IOCTL:キャラクタ・デバイスにコントロール・データを送る

IN: **AL** =**03h**
BX =ハンドル
CX =コントロール・データのバイト数
DS:DX =バッファのアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =転送されたバイト数（**CF=0**の場合）
=エラーコード（**CF=1**の場合）

IOCTL:ブロック・デバイスからコントロール・データを受け取る

IN: **AL** =**04h**
BL =ドライブ番号（**00h**=カレント，**01h=A:,02h=B:**…）
CX =コントロール・データのバイト数
DS:DX =バッファのアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =転送されたバイト数（**CF=0**の場合）
=エラーコード（**CF=1**の場合）

IOCTL:ブロック・デバイスにコントロール・データを送る

IN: **AL** =**05h**
BL =ドライブ番号（**00h**=カレント，**01h=A:,02h=B:**…）
CX =コントロール・データのバイト数
DS:DX =バッファのアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =転送されたバイト数（**CF=0**の場合）
=エラーコード

IOCTL:入力ステータスのチェック

IN: **AL** =**06h**
BX =ファイル・ハンドル番号

OUT: **CF** =**0**（正常）
=**1**（エラー）
AL =**FFh**（**CF=0**の場合．レディ状態）
=**00h**（**CF=0**の場合．**EOF**もしくはレディ状態でない）
AX =エラーコード（**CF=1**の場合）

IOCTL:出力ステータスのチェック

IN: **AL** =**07h**
BX =ファイル・ハンドル番号

OUT: **CF** =**0**（正常）
=**1**（エラー）
AL =**FFh**（**CF=0**の場合．レディ状態）
=**00h**（**CF=0**の場合．レディ状態でない）
AX =エラーコード（**CF=1**の場合）

備考: 出力がファイルの場合には常にレディ状態になる

IOCTL:ブロック・デバイスが交換可能か調べる

IN: **AL** =**08h**
BL =ドライブ番号（**00h**=カレント，**01h=A:**，**02h=B:**…）

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =**00h**（**CF=0**の場合．交換可能）
=**01h**（**CF=0**の場合．交換不可能）
=エラーコード（**CF=1**の場合）

IOCTL:リモート・ブロック・デバイスの検出

IN: **AL** =**09h**
BL =ドライブ番号（**00h**=カレント，**01h=A:**，**02h=B:**…）

OUT: **CF** =**0**（正常）
=**1**（エラー）
DX =デバイス属性ワード（**CF=0**の場合）
ビット**12** =**1** リモート（ネットワーク）
=**0** ローカル（直接接続）
AX =エラーコード（**CF=1**の場合）

備考: MS-Networksが必要．

IOCTL:リモート・ハンドルの検出

IN: **AL** =**0Ah**
BX =ファイルハンドル番号

OUT: **CF** =**0**（正常）
=**1**（エラー）
DX =ファイルまたはデバイス属性ワード
ビット**15**=1（リモート（ネットワーク））
=**0**（ローカル（直接接続））

備考: MS-Networksが必要.

IOCTL:リトライ回数の設定

IN: **AL** =**0Bh**
DX =リトライの回数
CX =リトライ間の待ち時間

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =エラーコード

IOCTL:キャラクタ・デバイス（プリンタ）のリトライ回数の設定／取得

設定

IN: **AL** =**0Ch**
CL =**45h**
BX =ファイル・ハンドル番号
CH =カテゴリー・コード（05h＝プリンタ）
DS:DX =データバッファのアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =エラーコード

取得

IN: **BX** =ファイル・ハンドル番号
CH =カテゴリー・コード（**05h**＝プリンタ）
DS:DX =データバッファのアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =エラーコード

備考: 「データバッファ」は1ワードでリトライ回数が入っている.

IOCTL:ブロック・デバイスのコントロール一般

パラメータ設定

IN: **AL** =**0Dh**
CL =**40h**
BL =デバイス番号（**00h**=カレント，**01h=A:**，**02h=B:**…）
CH =カテゴリー・コード（**08h**＝ディスクドライブ）
DS:DX =パラメータ・ブロックのアドレス

OUT: **CF** =0（正常）
=1（エラー）
AX =エラーコード

論理ドライブへのトラック書き込み

IN: **AL** =**0Dh**
CL =**41h**
BL =デバイス番号（**00h**=カレント，**01h=A:**，**02h=B:**…）
CH =カテゴリー・コード（08h＝ディスクドライブ）
DS:DX =パラメータ・ブロックのアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =エラーコード

論理ドライブのトラック，フォーマットとベリファイ

IN: **AL** =**0Dh**
CL =**42h**
BL =デバイス番号（**00h**=カレント，**01h=A:**，**02h=B:**…）
CH =カテゴリー・コード（**08h**＝ディスクドライブ）
DS:DX =パラメータ・ブロックのアドレス

OUT: **CF** =**0**（正常）
=**1**（エラーなし）
AX =エラーコード

ブロック・デバイスのパラメータ取得

IN: **AL** =**0Dh**
CL =**60h**
BL =デバイス番号（**00h**=カレント，**01h=A:**，**02h=B:**…）
CH =カテゴリー・コード（**08h**＝ディスクドライブ）
DS:DX =パラメータ・ブロックのアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =エラーコード

論理ドライブのトラック読みだし

IN: **AL** =**0Dh**
CL =**61h**
BL =デバイス番号（**00h**=カレント，**01h=A:**，**02h=B:**…）
CH =カテゴリー・コード（08h＝ディスクドライブ）
DS:DX =パラメータ・ブロックのアドレス

OUT: **CF** =0（正常）
=1（エラー）
AX =エラーコード

論理ドライブのトラック・ベリファイ

IN: **AL** =**0Dh**
CL =**61h**
BL =デバイス番号（**00h**=カレント，**01h=A:**，**02h=B:**…）
CH =カテゴリー・コード（**08h**＝ディスクドライブ）
DS:DX =パラメータ・ブロックのアドレス

OUT: **CF** =0（正常）
=1（エラー）
AX =エラーコード

IOCTL:論理ドライブ・マップの取得

IN: **AL** =**0Eh**
BX =ドライブ番号（**00h**=カレント，**01h=A:**，**02h=B:**…）

OUT: **CF** =0（正常）
AL =**00h** 1つだけアサインされている
=**(01**～**1Ah)**論理ドライブA～Zがアサイン
CF =1（エラー）
AX =エラーコード

備考: ブロック・デバイス（FDD）に複数の論理ドライブがアサインされているかのチェック．FDD1台のシステムで有効．

IOCTL:論理ドライブ・マップの設定

IN: **AL** =**0Fh**
BX =ドライブ番号（00h=カレント，01h=A:，02h=B:…）

OUT: **CF** =**0**（正常）
CF =**1**（エラー）
AX =エラーコード

備考: ブロック・デバイスに次の論理ドライブを設定する

IOCTL:サポートの問い合わせ（ハンドル）

IN: **AL** =**10h**
BX =ファイルハンドル番号
CH =カテゴリー・コード（**01h:**シリアル，**03h:**コンソール，
05h:パラレル・プリンタ）
CL =ファンクションコード
45h：繰返カウントの設定
65h：繰返カウントの取得

OUT: **CF** =0:サポートしている
CF =1:サポートしていない（エラー）
AX =エラーコード

IOCTL:サポートの問い合わせ（デバイス）

IN: **BX** =ドライブ番号（**00h**=カレント，**01h=A:**，**02h=B:**…）
CH =カテゴリー・コード（**08h**=ディスクドライブ）
CL =(**AL=0Dh**)で使用するマイナーコードと対応

OUT: **CF** =0:サポートしている
CF =1:サポートしていない（エラー）
AX =エラーコード

45h ファイルハンドルのコピー

IN: **BX** =オープンされているファイルハンドル番号

OUT: **CF** =0（正常）
=1（エラー）
AX =新しいファイルハンドル番号（**CF=0**の場合）
=エラーコード（**CF=1**の場合）

46h 指定したファイルハンドルへのコピー

IN: **BX** =既存のオープンハンドル
CX =新規のオープンハンドル

OUT: **CF** =0（正常）
=1（エラー）
AX =エラーコード（**CF=1**の場合）

47h カレント・ディレクトリの取得

IN: **DL** =ドライブ番号（**00h**=カレント，**01h=A:**，**02h=B:**…）
DS:SI =ディレクトリ名を入れるバッファのアドレス

OUT: **CF** =0（正常）
CF =1（エラー）
AX =エラーコード（**CF=1**の場合）

備考: ディレクトリ名を入れるバッファは64バイト．返ってきたディレクトリ名にドライブは含まれない．

48h メモリ・ブロックの割り当て

IN: **BX** =割り当てるメモリサイズ（パラグラフ数）

OUT: **CF** =0（正常）

CF =1（エラー）

AX =割り当てられたメモリのセグメント（**CF=0**の場合）

=エラーコード（**CF=1**の場合）

BX =割り当て可能なメモリ・サイズ（パラグラフ，**CF=1**の場合）

49h メモリ・ブロックの解放

IN: **ES** =解放するメモリのセグメント

OUT: **CF** =0（正常）

CF =1（エラー）

AX =エラーコード（**CF=1**の場合）

4Ah メモリ・ブロックのサイズ変更

IN: **BX** =新しいサイズ（パラグラフ数）

ES =割り当てられているメモリのセグメント

OUT: **CF** =0（正常）

CF =1（エラー）

AX =エラーコード（**CF=1**の場合）

BX =割り当て可能なメモリ・サイズ（パラグラフ，**CF=1**の場合）

4Bh プログラムのロード/実行

プログラムのロードと実行

IN: **AL** =**00h**

DS:DX =パス名（ASCIIZ文字列）のアドレス

ES:BX =パラメータ・ブロックのアドレス

OUT: **CF** =0（正常）

=1（エラー）

AX =エラーコード（**CF=1**の場合）

プログラムのロード（実行しない） ＊Ver5以上

IN: **AL** =**01h**

DS:DX =パス名（ASCIIZ文字列）のアドレス

ES:BX =パラメータ・ブロックのアドレス

OUT: **CF** =0（正常）

CF =1（エラー）

AX =エラーコード

備考: NECのマニュアルには記載されていない．

オーバーレイのロード

IN: **AL** =**03h**
DS:DX =パス名（ASCIIZ文字列）のアドレス
ES:BX =パラメータ・ブロックのアドレス

OUT: **CF** =0（正常）
CF =1（エラー）
AX =エラーコード

備考: **AL**=01hのときは**PSP**を作るが，**AL=03h**では**PSP**は作らない．

新しいプログラムの実行準備　＊Ver5以上

IN: **AL** =**05h**
DS:DX =**EXECSTATE**構造体のアドレス

OUT: なし

備考: **EXECSTATE**構造体の構造は次のとおり

esReserved	**dw**	?	渡される環境のセグメント
esFlags	**dw**	?	タイプ・フラグ
esProgName	**dd**	?	プログラム名のアドレス
esPSP	**dw**	?	プログラムのPSPのセグメント
esStartAddr	**dd**	?	プログラムのCS:IPの初期値
esProgSize	**dd**	?	PSPを含めたプログラム・サイズ

4Ch プロセスの終了

IN: **AL** =リターン・コード

OUT: なし

4Dh 子プロセスからリターン・コードを取得

IN: なし

OUT: **AH** =**00h**（通常の終了）
=**01h**（Ctrl-Cによる終了）
=**02h**（致命的エラーハンドラによる終了）
=**03h**（常駐終了）
AL =子プロセスから渡されたリターン・コード

4Eh 最初に一致するファイル名の検索

IN: **DS:DX** =パス名（ASCIIZ文字列）のアドレス
CX =属性

OUT: **CF** =**0**（ファイルが見つかった（DTAに情報がセットされる））
=**1**（ファイルが見つからない（エラー））
AX =エラーコード

4Fh 次に一致するファイル名の検索

IN: なし

OUT: **CF** =0（ファイルが見つかった（DTAに情報がセットされる））
CF =1（ファイルが見つからない（エラー））
AX =エラーコード

54h ベリファイ・フラグの取得

IN: なし

OUT: **AL** =**00h**（ベリファイはオフ）
=**01h**（ベリファイはオン）

56h ディレクトリ・エントリの変更

N: **DS:DX** =古いパス名（ASCIIZ文字列）のアドレス
ES:DI =新しいパス名のアドレス

OUT: **CF** =0（正常）
=1（エラー）
AX =エラーコード

備考: 隠しファイル，システム・ファイル，サブディレクトリは変更できない．

57h ファイルの日付，時刻の取得/設定

取得

IN: AL =00h
BX =ファイルハンドル番号

OUT: CF =0（正常）
=1（エラー）
CX =時刻（CF=0の場合）
DX =日付（CF=0の場合）
AX =エラーコード（CF=1の場合）

設定

IN: AL =01h
BX =ファイルハンドル番号
CX =設定する時刻
DX =設定する日付

OUT: CF =0（正常）
=1（エラー）
AX =エラーコード

58h アローケーション・ストラテジ（メモリ割り当て方法），上位メモリ・ブロックのリンク状態の取得/設定，

アロケーション・ストラテジの取得

IN: **AL** =**00h**

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =割り当て方法（**CF=0**の場合）
=エラーコード（**CF=1**の場合）

アロケーション・ストラテジの設定

IN: **AL** =**01h**
BX =割り当て方法コード

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =エラーコード

上位メモリ・ブロック（UMB）のリンク状態の取得

IN: **AL** =**02h**

OUT: **CF** =**0**（正常）
=**1**（エラー）
AL =**00h**（UMBがリンクされていない）
=**01h**（UMBがリンクされている）

上位メモリ・ブロックのリンク状態の設定

IN: **AL** =**03h**
BX =**00h**（UMBをリンクしない）
=**01h**（UUMBをリンクする）

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =エラーコード

59h 拡張エラーコードの取得

IN: **BX** =**00h**

OUT: **AX** =拡張エラーコード
BH =エラークラス
BL =推奨されるエラー対処
CH =エラーの発生箇所

備考: レジスタは破壊される

5Ah テンポラリ・ファイルの作成

IN: **CX** =ファイル属性
DS:DX =パス名（ASCIIZ文字列）のアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =ファイルハンドル番号
=エラーコード
DS:DX =パス名（ASCIIZ文字列）のアドレス

備考: パス名は'¥'で終わり，その後に13バイトの空きが必要

5Bh 新しいファイルの作成

IN: **CX** =ファイル属性
DS:DX =パス名（ASCIIZ文字列）のアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =ファイルハンドル番号（**CF=0**の場合）
=エラーコード（**CF=1**の場合）
DS:DX =パス名（ASCIIZ文字列）のアドレス

5Ch ファイルアクセスのロック/ロック解除

ロック

IN: **AL** =00h
BX =ファイルハンドル番号
CX:DX =ロックする領域の開始アドレス
（ファイルの先頭からのオフセット）
SI:DI =ロックする領域のサイズ

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =エラーコード

ロック解除

IN: **AL** =01h
BX =ファイルハンドル番号
CX:DX =ロックを解除する領域の開始アドレス
（ファイルの先頭からのオフセット）
SI:DI =ロックを解除する領域のサイズ

OUT: **CF** =**0**（正常）
=**1**（エラー）
AX =エラーコード

5Eh ネットワーク関係ファンクション

マシン名を取得する

IN: **AL** =**00h**
DS:DX =マシン名バッファ（16バイト）のアドレス

OUT: **CF** =**0**（正常）
=**1**（エラー）
CH =**00h**（マシン名無効）
=その他：マシン名有効
CL =マシンのNETBIOS番号
DS:DX =マシン名のアドレス
AX =エラーコード

プリンタセットアップ文字列の設定

IN: **AL** =**02h**
BX =割り当てリストのインデックス番号
CX =セットアップ文字列の長さ（最大64バイト）
DS:SI =セットアップ文字列のアドレス

OUT: **CF** =0（正常）
=1（エラー）
AX =エラーコード

プリンタセットアップ文字列の取得

IN: **AL** =**???**
BX =割り当てリストのインデックス番号
DS:SI =セットアップ文字列を得るバッファのアドレス

OUT: **CF** =0（正常）
=1（エラー）
CX =セットアップ文字列の長さ（最大６４バイト）
DS:SI =セットアップ文字列を得るバッファのアドレス
AX =エラーコード（**CF=1**の場合）

5Fh ネットワーク関連ファンクション

割り当てリストのエントリを取得

IN: **AL** **=02h**
BX =割り当てリストのインデックス
DS:SI =ローカル名のためのバッファ（16バイト）のアドレス
ES:DI =リモート名のためのバッファ（128バイト）のアドレス

OUT: **CF** **=0**（正常）
=1（エラー）
BH **=00h**（（有効なデバイス．**CF=0**の場合）
=01h（無効なデバイス．**CF=0**の場合）
BL **=03h**（プリンタ）
=04h（ドライブ）
CX =ユーザー変数域
AX =エラーコード（**CF=1**の場合）

割り当てリストのエントリの作成

IN: **AL** **=03h**
BL **=03h**（プリンタ）
=04h（ドライブ）
CX =ユーザー変数域
DS:SI =ソース・デバイス名のアドレス
ES:DI =ディスティネーション・デバイス名のアドレス

OUT: **CF** **=0**（正常）
=1（エラー）
AX =エラーコード

割り当てリストのエントリの取り消し

IN: AL **=04h**
DS:SI =デバイス名のアドレス

OUT: **CF** **=0**（正常）
=1（エラー）

62h ＰＳＰアドレスの取得

IN: なし

OUT: **BX** =カレント・プロセスのPSPのセグメントアドレス

67h オープン可能な最大ハンドル数の設定

IN: **BX** =設定する最大ハンドル数

OUT: **CF** =0（正常）
=1（エラー）

備考: DOS3.3以上で使用可能（NECの3.3は内部3.21のため使用不可）.

68h ファイルのコミット（5.0以降）

IN: **BX** =フラッシュするファイルのファイルハンドル番号

OUT: **CF** =0（正常）
=1（エラー）

6Ch ファイルの拡張オープン/作成

IN: **BX** =ファイルをオープンするときのモード
CX =ファイルのアトリビュート
DX =アクション
DS:SI =パス名のアドレス

OUT: **CF** =0（正常）
=1（エラー）

コード	意　味
01h	ファンクション・コードが無効
02h	ファイルが見つからない
03h	パス名が見つからない
04h	ファイルをオープンし過ぎている
05h	アクセスできない
06h	ハンドルが無効
07h	MCB（Memory Control Block）が破損
08h	メモリが足りない
09h	メモリブロック・アドレスが無効
0Ah	環境が無効
0Bh	書式が無効
0Ch	アクセス・コードが無効
0Dh	データが無効
0Eh	予約
0Fh	ドライブ名が無効
10h	カレント・ディレクトリを削除しようとした
11h	同じデバイスではない
12h	これ以上ファイルはない
13h	ディスクがライトプロテクトされている
14h	ディスク・ユニットが不良
15h	ドライブが準備できていない
16h	ディスクコマンドが無効
17h	CRCエラー
18h	長さが無効
19h	シークエラー
1Ah	MS-DOSのディスクではない
1Bh	セクタが見つからない
1Ch	紙切れ
1Dh	書き込みエラー
1Eh	読み出しエラー
1Fh	一般的なエラー
20h	共有違反
21h	ロック違反
22h	ディスクが不正
23h	FCB使用不可能
24～31h	予約
32h	ネットワーク・リクエストが未サポート
33h	リモート・コンピュータがLISTEN状態でない
34h	ネットワーク名が重複している
35h	ネットワーク名が見つからない
36h	ネットワーク・ビジー
37h	ネットワーク・デバイスはこれ以上ない
38h	ネットワークBIOSの限界を越えた
39h	ネットワーク・アダプタのハード・エラー

コード	意　味
3Ah	ネットワークから不正な応答があった
3Bh	予期できないネットワーク・エラー
3Ch	リモート・アダプタが不正
3Dh	プリント待ち行列が一杯
3Eh	待ち行列が一杯ではない
3Fh	プリント・ファイルのためのスペースが足りない
40h	ネットワーク名はすでに削除されている
41h	アクセス不可
42h	ネットワーク・デバイスのタイプが不正
43h	ネットワーク名が見つからない
44h	ネットワーク名の限界を越えた
45h	ネットワークBIOSセッションの限界を越えた
46h	一時休止
47h	ネットワークの要求が受け付けられない
48h	プリンタかディスクのリダイレクト休止
49～4Fh	予約
50h	ファイルが存在する
51h	予約
52h	作成不能
53h	割り込み24hのエラー
54h	ネットワーク構造が不正
55h	割り当てずみ
56h	パスワード無効
57h	パラメータ無効
58h	ネットワークへの書き込みエラー
5Ah	システム関連ファイルがロードされていない

添付ディスクの使い方

添付ディスクのバックアップ

添付ディスクには，『ZZ386ミニ版』とサンプル・プログラムが収録されています．添付ディスクは以下の手順でバックアップしてからお使いください．

ここでは，ドライブA，Bがフロッピーディスク・ドライブであると仮定しています．また，DOSのバージョンによっては，一部のメッセージが異なる場合があります．

①Aドライブに MS-DOSのシステム・ディスクを入れて，コンピュータを起動する．

②Bドライブに新しい2HDのディスクを入れる．

③新しいディスクをフォーマットする．

```
FORMAT B: /M
```

③「どれかキーを押してください」と表示されたら，RTEURN キーを押す．

④「別のディスクをフォーマットしますか」と表示されたら N キーを押す．

⑤ディスクをバックアップする．

```
DISKCOPY A: B:
```

⑥「準備ができたらどれかのキーを押してください」と表示されたらAドライブのディスクを添付ディスクと交換して，RTEURN キーを押す．

⑦「もう一度実行しますか」と表示されたら，N キーを押す．

『ZZ386ミニ版』のインストール

『ZZ386ミニ版』は以下の方法で他のディスクにインストールしてから使います．『ZZ386ミニ版』はフロッピーディスクでも使えますが，ハードディスクで使うことを想定しているので，ここではハードディスクをAドライブ，添付ディスクをBドライブとして説明します．

①Bドライブにコピーした添付ディスクを入れます．

②次のように入力して，インストール用バッチ・ファイルを起動します．

```
INSTZZ B A
```

③AドライブにZZ386というディレクトリが作られ，その中に『Z386ミニ版』とサンプル・プログラムがコピーされます．

『Z386ミニ版』の起動

カレント・ディレクトリをインストールしたディレクトリに移動して，

```
ZZ386
```

とタイプすると，ZZ386ミニ版が起動します．

詳しい使い方は，第4部「逆アセンブラ『ZZ386』を使う」を参照してください．

```
B:¥
|-INSTZZ   BAT     497 93-01-26
|-SAMPLE <DIR>--.       <DIR>         93-01-22
|               |-..      <DIR>         93-01-22
|               |-CAL       ASM    2895 92-11-18
|               |-CAL       COM     312 92-11-18
|               |-CLINE1    ASM    2020 92-11-18
|               |-CLINE1    COM      82 92-11-18
|               |-CLINE2    ASM    1317 92-11-18
|               |-CLINE2    COM      67 92-11-18
|               |-CLINE3    ASM    2296 92-11-18
|               |-CLINE3    COM     146 92-11-18
|               |-COM       BAT      29 92-01-23
|               |-DSIZE     ASM    7232 92-11-18
|               |-DSIZE     COM     867 92-11-18
|               |-EXE       BAT      33 92-02-10
|               |-FS        ASM   12651 92-11-18
|               |-FS        EXE    1747 92-11-18
|               |-S1        ZPJ     522 92-06-24
|               |-S2        BAT      39 92-01-30
|               |-S2        ZPJ     522 92-06-24
|               |-S3        BAT      68 92-01-30
|               |-S3        ZPJ     522 92-06-24
|               |-S3B       ZPJ     522 92-06-24
|               |-S4        BAT      65 92-01-30
|               |-S4X       EQU       5 92-01-31
|               |-S4X       EXE     569 92-01-31
|               |-S4X       LST    2881 92-01-31
|               |-S4X       MAP     557 92-01-31
|               |-S4X       ZPJ     522 92-06-24
|               |-S4XX      BAT      28 92-01-30
|               |-S4XX      ZPJ     522 92-06-24
|               |-SAMPLE1   ASM     276 92-06-25
|               |-SAMPLE1   COM      25 92-01-23
|               |-SAMPLE1   LST    1338 92-01-23
|               |-SAMPLE1   MAP      99 92-01-23
|               |-SAMPLE2   ASM     459 92-07-03
|               |-SAMPLE2   EXE     557 92-06-24
|               |-SAMPLE2   LST    3566 92-06-25
|               |-SAMPLE2   MAP      44 92-02-11
|               |-SAMPLE22  ASM     628 92-07-03
|               |-SAMPLE22  EXE     557 92-02-11
|               |-SAMPLE22  LST    2266 92-02-11
|               |-SAMPLE22  MAP     224 92-02-11
|               |-SAMPLE3   INC     927 92-01-30
|               |-SAMPLE31  ASM     357 92-07-03
|               |-SAMPLE31  COM      52 92-01-30
|               |-SAMPLE31  LST    3211 92-01-30
|               |-SAMPLE31  MAP     753 92-01-30
|               |-SAMPLE32  ASM     317 92-07-03
|               |-SAMPLE32  LST    3138 92-01-30
|               |-SAMPLE3B  COM      52 92-01-30
|               |-SAMPLE4   INC     758 92-01-30
|               |-SAMPLE41  ASM     426 92-07-03
|               |-SAMPLE41  EXE     569 92-01-30
|               |-SAMPLE41  LST    3194 92-01-30
|               |-SAMPLE41  MAP     987 92-01-30
|               |-SAMPLE42  ASM     337 92-07-03
```

```
|               |-SAMPLE42 LST    2859 92-01-30
|               |-TIMEA    ASM    2823 92-07-14
|               |-TIMEA    COM     152 92-06-25
|               |-TIMEA    MAP     103 92-02-18
|               |-TIMES    ASM    3662 92-07-03
|               |-TIMES    COM     270 92-06-25
|               |-TIMES    MAP     103 92-02-18
|               +-S4       ZPJ     522 93-03-29
+-ZZ386  <DIR>--.      <DIR>         93-01-22
                |-..     <DIR>         93-01-22
                |-ZZ386    EXE 271030 92-06-01
                |-COMENT98 I21    5498 91-10-06
                |-COMENT98 INT     797 91-10-07
                |-COMENTJ  PSP    1183 92-04-18
                |-8086     TBL   13308 91-11-14
                |-ZZ386    HLP 116428 92-02-26
                +-COMENT   EMM    1754 91-09-09
```

◎最後に
～もうあなたはアセンブラ・プログラマー

ここまで来ればあなたは「動くプログラム」を書けるはずです．あとはどんどん書くことが上達への早道です．

とりあえずプログラムが書ける状態になるようにという構成なので，ニーモニック，アセンブラ疑似命令，マクロ記述など超特急で走ってきました．

また，逆アセンブラ「ZZ386」を効率良く使って，「少しでも早くアセンブラ・プログラマーになってもらいたい」という構成なので，基本的なことしか述べていない箇所が随分とあります．

疑問を持った箇所などは使っているアセンブラのマニュアル，他のアセンブラ解説書，インテルのマニュアルなども合わせて参考にしてください．

またプログラムを書くうえでエディタは必要不可欠です．実際にプログラムを作成するときは複数のソースファイルなどを同時に編集するので「複数ファイルの編集機能」はぜひ必要な機能です．コンパイラなどに付属したエディタではどうしても機能が低く，プログラム作成の効率にも影響します．

エディタ単体として販売されているものであれば，まず機能的には問題ありません．誰かが「エディタ賛美は宗教です」という意味のことをいっていましたが，慣れの問題が1番だと思います．

PC-9801で使用できるエディタはたくさんありますが，どれが良いとかはとても言えません．私が使用しているのはMifes（メガソフト）とVzエディタ（ビレッジセンター）です．8:2くらいの比率で使っています．

参考書

お奨め文献と独断的な評価です．

1）ザ・８０８６ブック　　産業報知センター
ラッセル・アレクシー／ジョージ・アレクシー共著　　吉川敏則　訳
これはお奨めです．これがないとZZ386はできなかった？

2）ザ８０３８６ブック　　富士ソフトウェア
Ross P.Nelson著　富士ソフトウェア教育出版部訳・編
３８６本のはしり．

3）８０３８６　プログラミング　　工学社
John H.Crawford+Patrik P.Gelsinger　　訳／岩谷宏
岩谷さんの翻訳が好き．
コストパフォマンスが良い．（重量／値段？？）

4）MS-DOSエンサイクロペディア　Volume1　　アスキー出版局
プログラマのために，MS-DOSを理解するのに最高の本．
値段が高いのが欠点．（洋書と比べると高くない！）

5）Microsoft
MS-DOS
Programmer's Reference　　Microsoft PRESS
今まで非公開だったDOSの情報が満載．
MS-DOS Ver5ユーザー必携．
英文だけど比較的読みやすい．翻訳本が出るでしょう．

INDEX

命令索引

-A-
AAA, …… 32
ANSI.SYS, ……26
AUTOEXEC.BAT, …… 21

-B-
BIOS, …… 11
BREAK, …… 22
BUFFERS, …… 22

-C-
CALL命令, …… 43
CODE, …… 57
COMSPEC, …… 25
COMファイル, ……56,96
CONFIG.SYS, …… 21
CPUの直行性, …… 29

-D-
DAA, …… 39
DATA, …… 57
DEVICE, …… 24
DEVICEHIGH, …… 24
DOS=, …… 23
DOSKEY.COM, …… 25

-E-
EMM386.EXE, …… 23
ENDIF, …… 86
EXE
　～ヘッダ, …… 58
EXEファイル,……56,96
EXTRN, ……81

-F-
FILES,…… 22

-G-
GLOBAL, …… 82
-
H-
HIMEM.SYS, …… 23

-I-
IFB, …… 86
IFDIF, …… 86
IFIDN, …… 86
IFNB, …… 86
int
　～サービス, ……12
-J-
JMP命令, …… 44

-L-
LABEL命令,…… 82
LASTDRIVE, …… 24
LOADHIGH, …… 25
LOCAL, ……79

-M-
MOUSE.SYS, …… 24

-P-
PRINT.SYS, …… 24
PSP, …… 61
PUBLIC, …… 80

-R-
RAMDISK.SYS,…… 24
RET命令, …… 43
RSDRV.SYS, ……24

-S-
SEGMENT, …… 61,64
SHELL, …… 23
SMARTDRV, …… 24
STACK, …… 65

-T-
TEST命令, ……37

索引

-あ-
アセンブラ, …… 20
　～疑似命令, …… 55
　～命令, …… 56
アセンブル, …… 20
アドレッシング・モード, …… 15,30
エイリアス, …… 79
オプション
　コマンドライン・～, ……159

-か-
加算命令, …… 31
減算命令, …… 33

-さ-
サーチ, ……122
算術演算命令, …… 31
算術シフト, …… 53
システム・コール, …… 10
自動解析, ……102
シフト，ローテート命令, …… 49
ジャンプ, ……116
　イミディエイト・～, …… 44
　オペランドへの～, ……117

クロスタグへ～, ……118
サーチタグへ～, ……118
～命令, …… 42
条件～, 45
絶対値への～, ……117
バック・～, ……117
前の位置へ～, ……117
メモリ間接～, …… 44
ユーザータグへ～, ……118
ラベル参照位置への～, ……118
レジスタ間接～, …… 44
終了
ZZ386の～, …… 105
常駐～, …… 164
プログラムの～, …… 163
乗算命令, …… 34
常駐型プログラム, …… 149
除算命令, …… 34
スタートアップ・ルーチン, …… 62
ストラテジ・ルーチン, …… 146
ストリング・プリミティブ命令, …… 37
ストリング命令, …… 37
セグメント
～・プリフィックス, …… 31
ソース, …… 29

-た-
ターゲット
～CPU, …… 95
～・ファイル, …… 92
タグ
クロス・～, …… 118
サーチ・～, …… 118
サーチ・～のセット, …… 124
ユーザー・～, …… 118
直接アドレッシング, …… 30
ディレクティブ, …… 56
数式，定義の～, …… 79
データのコピー命令, …… 29
デスティネーション, …… 29
デバイス
～・ヘッダ, ……146
デバイス・ドライバ, ……146

-な-
ニモニック, …… 28
入出力命令, …… 48

-は-
排他的論理和命令, …… 36
比較命令, …… 34
比較命令の方向, …… 39
否定命令, …… 35
ファイル
インクルード・～, …… 86
オブジェクト・～, …… 20
ソース・～, …… 20
ターゲット・～, …… 92
～のタイプ, …… 92
プロジェクト・～, …… 92
マップ・～, …… 60
フラグ
オーバーフロー・～, …… 47
キャリー・～, …… 47
サイン・～, …… 47
算術演算, …… 47
ゼロ・～, …… 47
ディレクション・～, …… 38
～に影響を与える命令, …… 47
補助キャリー・～, …… 47
プログラムの初期化, ……158
プロシージャ, …… 77
プロセッサ制御命令, …… 48
ヘッダ
EXE, ……142
デバイス・～, ……146
リクエスト・～, ……147
ヘルプ, ……106

-ま-
マクロ, …… 84
メイク, ……102

-ら-
ラベル, …… 80
ローカル・～, …… 78
リピート・プリフィックス命令, …… 40
リロケーション・データ, ……143
リンカ, …… 21
リンク, …… 20
ループ命令, …… 48
レジスタ
インデックス・～, …… 14
セグメント・～, …… 13,29
汎用～, …… 13
～間接アドレッシング, …… 30
ロード・モジュール, …… 59
論理演算命令, …… 34
論理シフト, …… 52
論理積命令, …… 34
論理和命令, …… 35

-わ-
割り込み
～サービス, …… 12
～命令, …… 49
～ルーチン, ……147

《著者略歴》

出海　勝富（でうみ　まさとみ）

1948年　兵庫県相生市で生まれる．

1967年　芝浦工業大学電子工学科中退．

1988年　（株）マイクロ・オーグ設立．代表取締役．

*

最初に手に入れたapple2がコンピュータ業界で仕事をするきっかけとなる．以後，ハードウェア，ソフトウェアを問わず，数々のシステム設計・開発に携わる．

*

波の音の聞こえる海の近くでプログラミングできれば独創的なものが作れるのでは…と思いつつ，現在Cコンパイラを企画中．

本書にミニ版が添付されている逆アセンブラ『ZZ386』をはじめ，8ビット逆アセンブラ『ZZ』，8ビット向けクロスアセンブラ『ZASM』，ファイル管理ソフト『MF』などを開発．

初心者が短時間で上達するための　86アセンブラ入門

平成5年8月5日初版発行　©1993

著　者　出海　勝富

発行人　星　正明

発行所　株式会社　工学社

〒151　東京都渋谷区代々木1-37-1　ぜんらくビル

電　話　(03) 3375-5784（代）［営業］

(03) 3320-1218（代）［編集］

振替口座　東京5-22510

定価2500円（本体2427円）

印刷：（株）耕文社

雑誌61474-04

I/O 別冊 86Assembler

初心者が短時間で上達するための
86アセンブラ入門

mini ZZ386

PC-9801用逆アセンブラ

- 5インチ・3.5インチ
  2HD・FD各1枚
- 対応機種：PC-9801
- MS-DOS3.1以上

maxell
MINI-FLOPPY DISK
SUPER RDII
Super Reliable & Durable Tribo tech

関連図書

コンピュータがわからない人のための
8086マシン語入門

数学は苦手，コンピュータはどうも…．そんな人のための8086マシン語超入門書

佐藤尚著　A5判248頁　定価1957円

80386プログラミング

Intelの開発スタッフが80386，80387のプログラミングを解説．エンジニア，システム設計者の必携書．

ジョン・H・クロフォード他著　岩谷宏訳

B5判688頁　定価3605円

86系演算サブルーチン

指数，対数，三角関数などの各種関数演算を網羅したサブルーチン集．

PC-9801上でのMASMを使ったアプリケーション開発手順解説も収録．

成田福雄著　A5判約424頁　定価2800円

86 Assembler

初心者が短時間で上達するための
86アセンブラ入門

STEP 1　最低限の知識を身につける

↓

STEP 2　他人のプログラムを読む

↓

STEP 3　自分でプログラムを書く

KOHGAKU-SHA

T1061474042506

定価2500円（本体2427円）雑誌61474-04